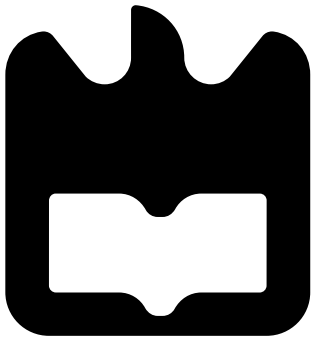
Universidade de Aveiro
Department of
Electronics, Telecommunications and Informatics
2015

João Carlos
Peralta Moreira

Simulador para Processadores de Sinal Digital de Arquitectura VLIW
An Instruction Set Simulator for VLIW DSP Architectures



João Carlos Peralta Moreira

Simulador para Processadores de Sinal Digital de Arquitectura VLIW
An Instruction Set Simulator for VLIW DSP Architectures

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Professor Doutor Manuel Bernardo Salvador Cunha, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e Doutor Mohamed Bamakhrama, Hardware Tools Engineer na equipa "Processor and Compiler Tools" no grupo "Imaging and Camera Technologies", Intel Eindhoven, Países Baixos

Dissertation presented to Universidade de Aveiro with the goal of achieving a Master's Degree in Electronics and Telecommunications, made with the scientific orientation of Professor Manuel Bernardo Salvador Cunha PhD, Professor at the Department of Electronic, Telecommunications and Informatics from Universidade de Aveiro and Mohamed Bamakhrama, Hardware Tools Engineer at Processor and Compiler Tools Team of Intel's Imaging and Camera Technologies Group, Eindhoven.

o júri / the jury

presidente / president

Professor Doutor Tomás António Mendes Oliveira e Silva
Professor Associado da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)

vogais / examiners committee

Professor Doutor João Manuel Cardoso
Professor Associado da Faculdade de Engenharia da Universidade do Porto

Doutor Mohamed Ahmed Mohamed Bamakhrama
Hardware Tools Engineer na equipa "Processor and Compiler Tools" em Intel Eindhoven, Países Baixos (co-orientador)

Agradecimentos / Acknowledgements

Aos meus pais, por todo o amor e sacrifício.

Special thanks to Felipe Chies and Mohamed Bamakhrama for their endless patience and support during the technical development of this work and to Orlando Moreira for his help on significantly improving the quality of this document. A final thanks to Menno Lindwer for the opportunity to work with his team and to Bernardo Cunha for his help and academic advisement.

Resumo

O grupo Intel Imaging and Camera Technologies (ICG) desenvolve soluções completas de processamento de imagem, utilizadas em várias plataformas móveis da mesma companhia. Para desenvolver novas arquitecturas, o grupo recorre a uma conjunto de ferramentas proprietário, construído no topo de uma linguagem de descrição de processadores VLIW. As ferramentas disponíveis neste conjunto incluem, entre outras, um gerador de código de hardware sintetisável, um compilador de ANSI C e um simulador.
Devido às suas características, o esquema de simulação actualmente utilizado não valida por completo binários gerados pelo compilador. Este trabalho apresenta uma solução baseada em interpretação direta de instruções com vista a colmatar essa falha. A solução aqui apresentada lê a descrição de um processador e gera o código fonte em SystemC de um simulador para essa mesma arquitectura. O código gerado é compilado e usado para validar a saída do compilador.
Os resultados finais em termos de performance superaram as expectativas para processadores de tamanho pequeno e médio, mas esta solução mostrou que não é tão escalável quanto a actualmente utilizada pelo grupo. Em suma, a solução desenvolvida mostrou apresentar um grau de escalabilidade inferior à solução actualmente utilizada pelo grupo.

Abstract

Intel's Imaging and Camera Technologies (ICG) develops complete image processing solutions, used in several mobile platforms from this company. To develop new architectures, ICG uses a proprietary tool set built on top of a VLIW Architecture Description Language. Some of the tools provided in this set are an Hardware Synthesizable code generator, an ANSI-C compiler and a Simulator.
Because the current simulation framework cannot validate binary code, the herein presented work provides a new interpretative simulation framework in order to allow the full validation of the compiler's binary output files. This solution uses the processor description to generate the source code for an architecture-specific SystemC Interpretative Instruction Set Simulator that can later be compiled and used to validate said binaries.
The final performance results of the presented solution exceed expectations for small and medium VLIW architectures, but also show that the presented framework does not scale as well as the current simulation framework for considerably large architectures.

Contents

List of Figures

List of Tables

Acronyms

ASIC Application Specific Integrated Circuit. , 15, *See also* ASIC
ASIP Application Specific Instruction-set Processor. , 15, *See also* ASIP

EDA Electronic Design Automation. , *See also* EDA

HDL Hardware Description Language. , *See also* HDL

ICG Imaging and Camera Technologies Group. , 1, 4, 9, 11, 12, 14–17, 20, 36, *See also* ICG
ILP Instruction Level Parallelism. , 2, 9, 12, *See also* ILP
Instruction Set Simulator Instruction Set Simulator. , 13, 16, 18–22, 28, 29, 35, 36, 41–44, *See also* ISS
IISS Interpretative Instruction Set Simulator. , 3, 4, 16–22, 24, 28, 29, *See also* IISS
ISA Instruction Set Architecture. , 4, 14, 17, 19, 22, 36, *See also* ISA

OoOE Out-of-Order Execution. , 8, 9, *See also* OoOE
OSCI Open SystemC Initiative. , *See also* OSCI

PAT Processor Architecture Template. , 9–11, 15, 18–21, 23, 25, 26, 28, 29, 31, 36, 38, 46, *See also* PAT
PEG Platform Engineering Group. , 1, *See also* PEG

RISC Reduced Instruction Set Computer. , 36, *See also* RISC
RPV Random Program Verification. , 4, 13, 14, 17, 18, *See also* RPV
RTL Register Transfer Level. , 1, 4, 11, 13, 15, 17–19, *See also* RTL

SDK Software Development Kit. , 1, *See also* SDK
SoC System on Chip. , 1, 9, *See also* SoC

VLIW Very Long Instruction Word. , 1, 4, 7–12, 14, 16, 17, 20, *See also* VLIW

Glossary

Application Specific Instruction-set Processor is a kind of processor designed with an instruction set specially tailored for a specific application or set of applications. It is usually used as a balanced trade-off between the the performance of an ASIC and the flexibility of a general purpose processor. , 15

Application Specific Integrated Circuit is an integrated circuit customized for a specific application. ASICs are usually a power efficient solution but have almost all no flexibility, usually making them only useful for the application they were designed to in the first place.

Electronic Design Automation is a category of tools used to accelerate integrated circuit development. Among the most known EDA tools one can find *logic synthesis*, *place and route*, *circuit simulators* and *verification*,. Because of the business value of this kind of software, EDA is nowadays also a big industry, led in sales by Synopsis.

Function Unit is a fully customizable Execution Unit with N inputs, N inputs and N operations. These operations can be either Load/Store operations, Arithmetic operations, Logic operations or a mixture of all the previous. The Function Unit can also contain vector operations, in which case it becomes a Vector Function Unit.

Hardware Description Language is a language used to program the structure and behaviour of electronic circuits. The two most widely used and known HDL languages are VHDL and Verilog.

High Level Synthesis is an automated design process that allows the generation of hardware logic based on a high-level programming language, like C or SystemC. The result hardware logic should, by definition, implement the same behaviour the code tries to describe.

Imaging and Camera Technologies Group is the group inside Intel Corporation responsible for developing complete solutions for imaging and video applications. It is also a sub-group of PEG, the Platform Engineering Group. , 1, 4, 9, 11, 12, 14–17, 20, 36

Instruction Level Parallelism is a measure of how many instructions of a program can be ran in parallel (without changing the result of the program). , 2, 9

Instruction Set Architecture also used as only Instruction Set is the set of all components of any computer architecture related to programming. This include not only the machine language (also known as opcodes), but also the native data types, addressing modes, register files, datapath and so on. , 4, 14, 17, 19, 22, 36

Instruction Set Simulator is a simulation model designed to target a certain processor architecture. This should mimic the processor behaviour as best as possible. , 4, 16–22, 24, 28

Compiled Instruction Set Simulator is a fast Instruction Set simulation model, designed to speed up simulation considerably when compared to an Interpretative ISS. In a compiled simulator, the target code is interpreted and a simulation model is created for that specific target code running on a specific target architecture. This way the overhead of decoding instructions disappears at the trade of not being able to change it during run-time (a change on target code requires a re-compilation of the entire simulation model). , 2, 3, 12, 17, 18, 21

Interpretative Instruction Set Simulator is an type of Instruction Set Simulator that mimics the architecture by directly interpreting (decoding and executing) a program binary. , 3, 21, 22, 29

Issue Slot is, in the most simple way, a limited set of Function Units. It can also contain N input and N output ports and it's main distinctive feature from the Function Unit is that it should be able to execute one (or start one) new operation every cycle. , 36, 38, 43

Lookup Table is a known program execution optimizer for repetitive (or very complex) operations. Beforehand (or in runtime), the program fills an array of pre-calculated values for a given kind of operation. After doing so, the program can simply get the calculated value for some known input (usually used also as index) from the array.

Open SystemC Initiative was the original independent organization responsible for promoting SystemC as an industry standard. It was recently merged with Accelera and became, as it is known nowadays, Accelera Systems Iniciative.

Out-of-Order Execution is an execution paradigm used in modern high-performance computer architectures. Like the name points out, in Out-of-Order architectures, the order by which operations are executed is not necessarily the same order by which they were fetched. The concept is to execute an operation as soon as its operands are available instead of when its the operation turn. This kind of paradigm speeds up the execution of a program, but on the other hand involves significant amounts of complicated control hardware to ensure that by changing the order some operation is executed does not compromise the final result of the program. , 8, 9

Platform Engineering Group is the group responsible for designing the silicon and platforms across all Intel's product segments.

Processor Architecture Template is the basic concept underlying the ICG VLIW Technology. It defines a set of parametrizable aspects of the VLIW architecture, allowing the design of a practically infinite set of VLIW architectures, starting from a simple high-level description. , 1, 9, 10, 15, 18–21, 23, 25, 26, 28, 29, 31, 36, 38, 46

Random Program Verification . , 4, 13, 14, 17, 18

Reduced Instruction Set Computer is a kind of computer architecture characterized by only having very simple instructions. Complex operations can run in this architecture but need to be split into smaller ones first, taking several cycles to execute. This might look like a disadvantage, but from the hardware perspective the architecture is much more simple and faster clock speeds may be employed. , 36

Register Transfer Level is a design abstraction level used in digital circuit design where a digital circuit is modeled in terms of the flow of signals between hardware registers and the logic operations performed in that flow. , 1, 4, 11, 13, 15, 17–19

Software Development Kit is a set of tools that allow the development of software to a certain architecture. Common tools inside a complete SDK are a compiler, a simulator and a debugger. , 1

System on Chip is an integrated circuit that incorporates all the components of an electric system inside a single die. This concept, very common in the embedded world, allows smaller power consumption and smaller implementation area, as the connections between all the components are also done inside the chip. , 1, 9

SystemC is, to a certain extent, a C++ class library with support for HDL-like variables. However, it is now considered by many as a new language and nowadays its features allow very accurate simulations and even some vendors already provide tools to use it as a source for high-level synthesis. SystemC became a IEEE Standard in 2005. , 12, 14

Very Long Instruction Word is a Computer Architecture designed to take explicit advantage of Instruction Level Parallelism. The main characteristic of these architectures is the ability to encode in the same instruction word several operations meant to be executed at the same time. This leads to bigger instruction word lengths (hence the name) but, if the program is properly scheduled, it allows extremely efficient execution due to reduced control overhead compared to other architectures. . , 1, 4, 7–12, 14, 16, 17, 20

Chapter 1

Introduction

1.1 Context

Nowadays, consumer smart phone cameras, specially in the high-end segment, are already capable of providing images with sizes of millions of pixels. However, the trend does not seem to stop there and vendors are already working towards higher computational capacity of digital camera processing units to accommodate the increase of image output requirements. This is the environment where Intel's Imaging and Camera Technologies Group (*ICG*) fits.

As a sub-group of the Platform Engineering Group (*PEG*), *ICG* is responsible for providing complete out-of-shelf imaging solutions to integrate in Systems on Chip (*SoC*). These System on Chip (*SoC*) are then used in high-end consumer products, like smart phones or tablets.

Due to the high demands on computational power, while restrained by energy consumptions, most common imaging solutions rely on fixed imaging hardware pipelines. Although power efficient, fixed hardware imaging pipelines lack flexibility and usually imply higher time-to-market and non-recurring engineering effort. Also, this approach do not allow late binding of the latest computational photography algorithms, as every processing step is *hard-coded* in the pipeline.

To address this issue, instead of a complete hardware pipeline, *ICG*'s approach relies on multi-core Very Long Instruction Word (*VLIW*) architectures, mixed with smaller fixed hardware blocks. This approach provides a balanced trade-off between flexibility and power consumption.

To develop these *VLIW* cores and software/firmware code to them, a complete tool chain exists that drives the development from a high level processor description. This tool chain allows the automatic generation of synthesizable Register Transfer Level (*RTL*) code and a Software Development Kit (*SDK*) for an architecture.

Figure 1.1 shows a representation of *ICG*'s main development flow. The flow has two user inputs: the *Platform Design* and the *Application Design*. The platform design comprises all the design specifications of the target platform. The application design comprises the algorithms and kernels to target the platform.

From the Platform Design input, the tool flow can generate synthesizable *RTL* and the platform description files for the software development tools, such as the Compiler and Simulator. The user is responsible for mapping the application to the different elements of the

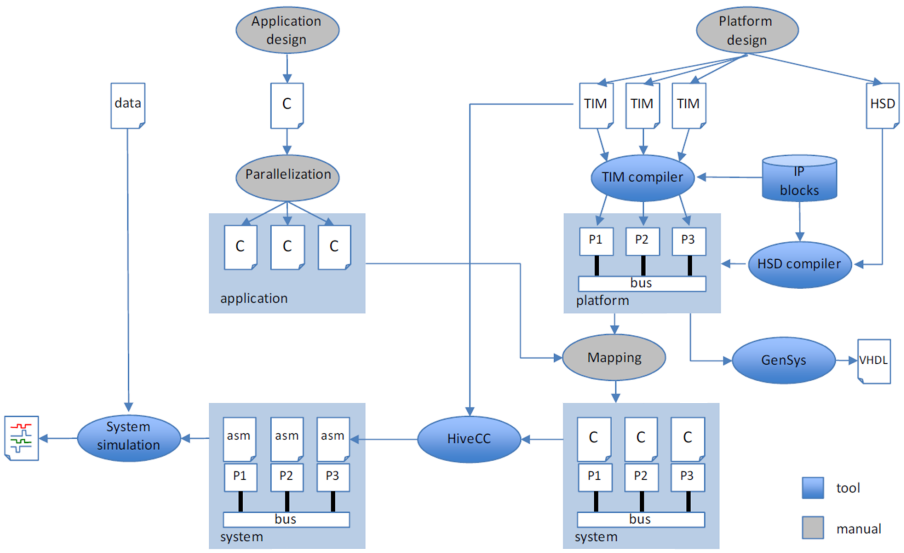


Figure 1.1: The complete VLIW processor development tool flow. Source: [1]

platform.

The retargetable compiler of the tool flow is called *HiveCC* and uses the mapped Application Design to target the platform architectures. A more detailed explanation on the tool flow can be found in Section 2.2.2.

The existing simulation framework lacks certain simulation functionalities needed by both firmware and hardware developers. These missing functionalities are presented in Section sec:motivation and later described in more detail in Section 4.1. This work presents a new simulation framework to address these needs.

1.2 Motivation

HiveCC comprises several sub-tools, among them a *front end* and an aggressive optimizing *scheduler*. In Figure 1.2 we can observe the full flow of *HiveCC*. With HiveCC, a *C*-source code is processed by a *front end* which generates an internal representation of the target program.

The internal representation is then sent to the *scheduler*, where it is heavily optimized and scheduled to explore the maximum of Instruction Level Parallelism (*ILP*) (targeting some specific architecture). The output of the *scheduler* is written in the form of a scheduled data-flow file.

From the scheduled code, the assembler generates a target executable binary to run on the target processor.

The tool flow also enables to simulate target code. Currently this is done using a static *Compiled Instruction Set Simulator* technique. The *Compiled Simulator* flow (shown in Figure 1.3) uses the scheduled code *.hive.s*, together with a simulation library, to generate a compiled simulation model of the target source code.

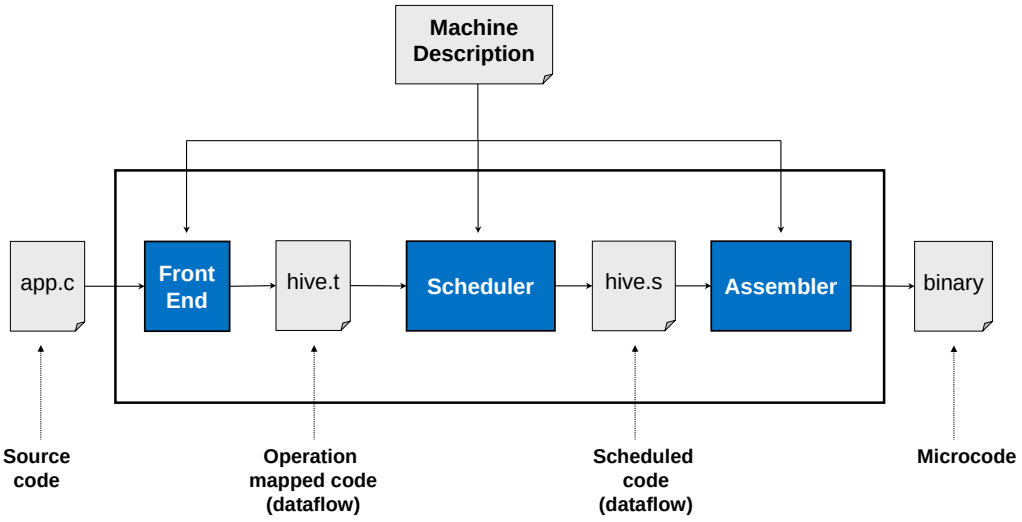


Figure 1.2: The compiler flow. Source: [1]

Because this is done using a static *Compiled Simulator* technique, the generated simulation model is unique for every target source code. This means that recompilation of the simulation model is required every time the developer makes a new change in the target source code being developed. On the other hand, *compiled simulation* techniques are known to provide considerable faster simulation speeds compared to *Interpretative Instruction Set Simulation* techniques [6].

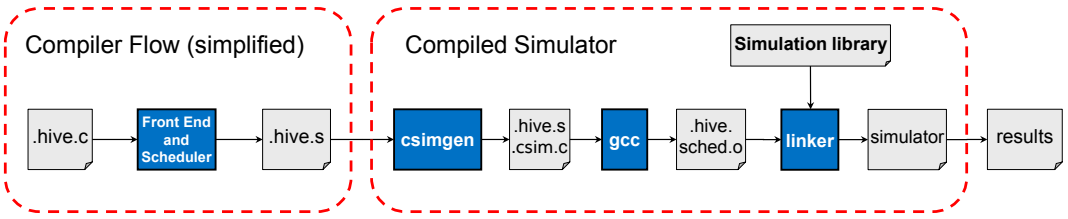


Figure 1.3: The compiled simulator flow Source: [1]

Although the compiled simulator approach typically provides shorter simulation times, the input for the entire simulation environment is the *intermediate representation* (Figure 1.3) code, not the target binary executable. This means room for error (and possibly very hard to find bugs) exists within the assembler path.

Figure 1.4 shows both the *Compiled Simulator* and Interpretative Instruction Set Simulator (*IISS*) approaches and how their flows compare to each other. The *Front End* and *Scheduler* start by generating an intermediate representation file (*.hive.s*) with the target program mapped and scheduled to the target architecture.

The *Compiled Simulator* approach then takes the scheduled code and generates an executable simulation model from it. The *Interpretative Instruction Set Simulator* approach uses

a fixed simulation model to interpret and simulate the target binary executable produced by the assembler.

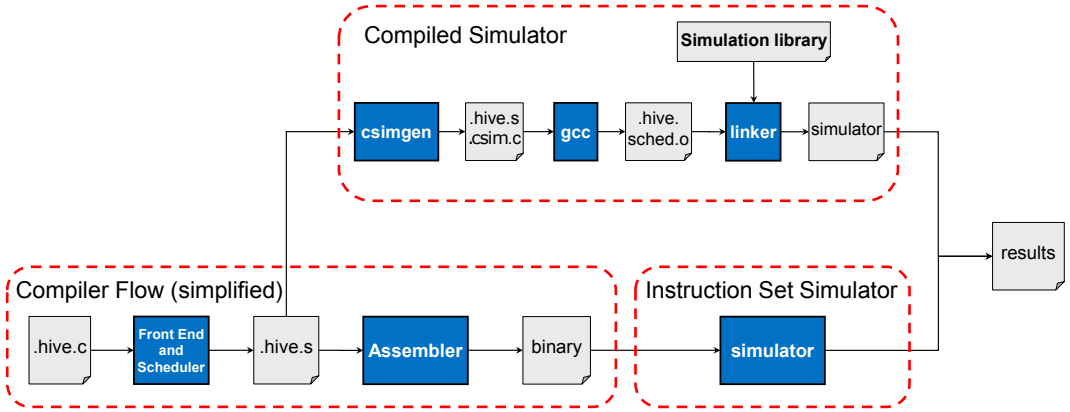


Figure 1.4: Comparison between Compiled Simulator and Interpretative Instruction Set Simulator complete software development flow (tools are marked with a blue background and output/exchange files are marked with a grey background)

Another reason for creating an *IISS* for *ICG VLIW* architectures has to do with enabling flexible Random Program Verification (*RPV*) for validation engineers.

RPV is a known processor validation technique [7], where random programs are executed in both the high level simulation model (*IISS*) and the low level *RTL* model, in order to find inconsistencies between them. If an inconsistency is found, it generally represents a bug in the *RTL* model.

The existing compiled simulator technique could be used to do this, with some adaptations, however full flexibility to validate the processor's Instruction Set Architecture (*ISA*) would be very hard to achieve. With the current approach engineers can only generate random *intermediate representation*, which still needs to go through the *assembler*, and may not allow full coverage of the *ISA*'s corner cases.

The last motivation to develop an *IISS* is to enable self-modifiable code. The static compiled simulation technique compiles a simulation model for a target application and does not support run-time changes to the program. An *IISS* can easily support such a feature, as instruction words are interpreted cycle by cycle.

1.3 Contributions

This work presents a new SystemC retargetable Interpretative Simulation Framework for *ICG VLIW* architectures. This new simulation framework is an attempt to provide an efficient binary validation mechanism for ICG's software/compiler flow. Current simulation techniques provide good coverage of ICG's processor functionality but do not validate target binary executables. Another limitation of the current simulation framework is not being able to simulate run-time changes to the program memory, either accidental or intentional. The presented solution also addresses this issue.

The presented simulation framework is composed by a Simulation Model template and a Simulation Model generator. The Simulation Model template describes a generic SystemC source code structure capable of modelling any processor from ICG's design space. The Simulation Model generator implements the Simulation Model template. This generator accepts a processor description file from ICG's current tool flow and generates a specific instance of the Simulation Model template able to simulate the given processor.

This document includes a description of the Simulation Model template and also presents some implementation details of the Simulation Model generator.

1.4 Outline

The outline of this document is as follows:

Chapter 2 - Background
: The Background presents a summary on concepts necessary for the full understanding of the work presented on this document. Section 2.1 discusses the VLIW architecture in general, its pros and cons and most known applications. Section 2.2 discusses the processor technology used by ICG to develop new Imaging Solutions. The last Section of this Chapter, Section 2.3 presents an overview on this technology's main features as it was used as a base for the new processor simulation framework.

Chapter 3 - Related Work
: The Related Work chapter presents a brief summary of work done previously by other authors on similar subjects. This chapter includes references on Retargetable tool chains, on Instruction Set Simulation, on VLIW architecture simulation and on improving simulation performance.

Chapter 4 - Problem Definition
: In the problem definition a more in-depth discussion about the motivation for this work is presented. In this chapter, the requirements for a solution to ICG's current simulation needs are presented. The chapter also discusses the challenges that such requirements represent.

Chapter 5 - Proposed Solution
: Given the requirements in Chapter 5, this chapter presents a proposal for a Simulation Framework that fulfills such requirements in order to address the presented motivation. This chapter is divided into two main Sections: the first Section presents a simulation model template, able to address the challenge of a Retargetable Simulation Framework; the second Section presents the structure of a code-generation tool capable of generating an instance of the previous template specific to a certain processor.

Chapter 6 - Evaluation
: The Evaluation Chapter first presents the Evaluation methodology used to evaluate the implementation of the proposed solution but also includes gathered data and the discussion and interpretation of the results.

Chapter 7 - Conclusions and Future Work
: The last Chapter is divided into two parts. The first part, the Conclusions, briefly

discuss one last time the motivation for this work, how the motivation lead to the requirements and to what extent the requirements were met in the end. The second part, the Future Work, presents an overview of what can still be done about this subject and some suggestions for next steps into a better solution.

Chapter 2

Background

2.1 VLIW architecture

VLIW architectures were first proposed in [8] by Joseph Fisher and his research group of Yale University. The concept behind *VLIW* relies on being able to put in a single instruction word many operations, independent from each other, capable of being executed in parallel. Quoting the same paper: "In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream".

VLIW cores have several execution units that can execute operations independently from each other. The units are usually connected with independent data-paths to the register (or registers) such that all can fetch and write back data simultaneously. Thus, each cycle, the architecture can start one operation per execution unit.

2.1.1 Clustered VLIW architectures

There are two main approaches to *VLIW* processor design: the single-register *VLIW* (NXP's TriMedia[9]) and the clustered *VLIW* (Texas Instruments' TMS320C62x [10]). The main difference between the two approaches is in the register file and datapath organization.

The single-register approach has all the data in the same register file and has the advantage that all execution units can access all registers at any time. However, because every execution unit requires at least three ports (two read and one write) from the register file, the amount of ports of the unified register file grows proportionally with the amount of execution units it has.

As the complexity of the implementation of a multi-ported register file increases with the number of ports, single-register architectures usually are limited to a certain amount of execution units (otherwise performance requirements, like clock speed, might not be able to be met).

To solve the many-ported register file complexity problem, the *VLIW* clustered architecture approach distributes the registers among several register files. In this approach, each smaller register file has less ports and can be implemented in hardware with less gates and faster clock cycles. To solve the inter-register file connectivity problem architectures implement alternative datapaths that can connect one cluster to another.

Figure 2.1 shows an example of a clustered *VLIW* architecture. This architecture (Texas Instruments' *TMS320C62x*) has 8 execution units and 2 register files distributed by 2 clusters. In each cluster there are 4 fully connected execution units to 1 register file and to enable communication between clusters, a single output port of each register file connects to the opposing cluster.

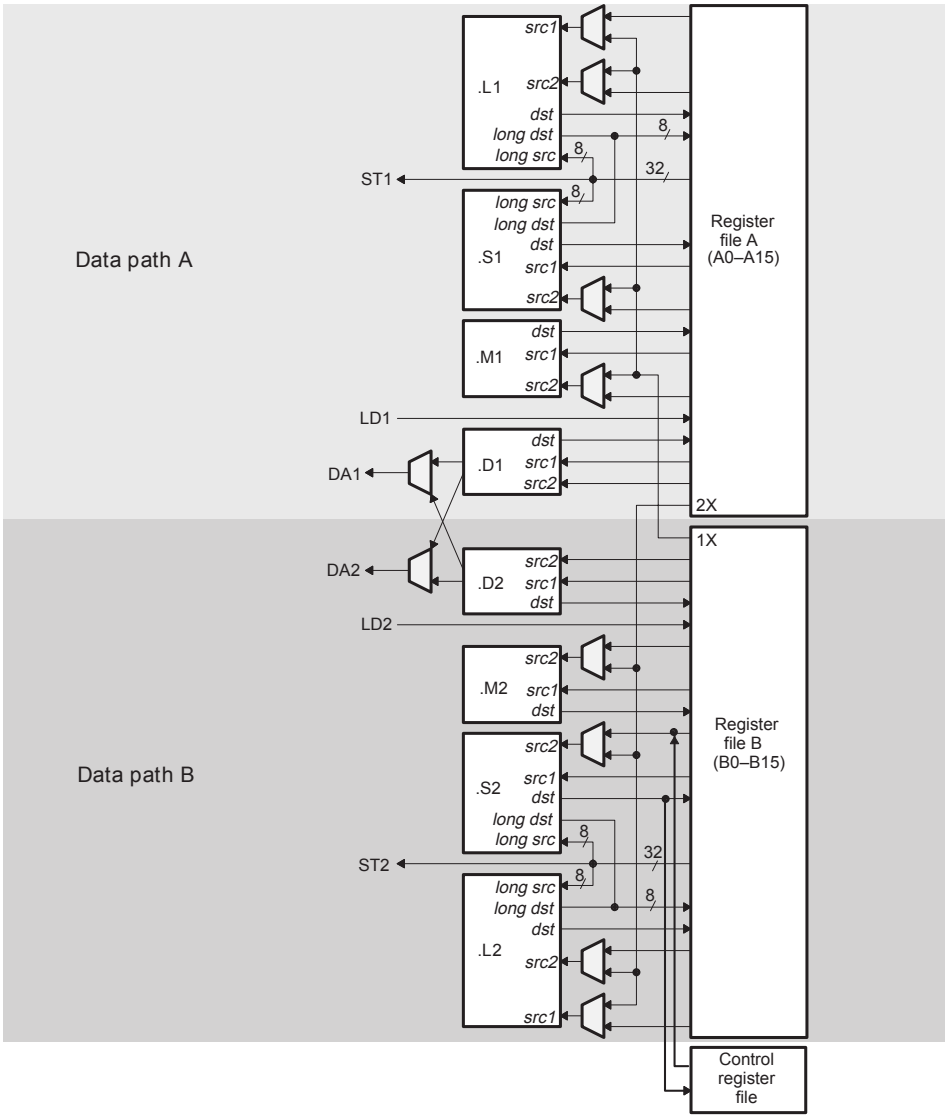


Figure 2.1: Example of a commercial clustered VLIW architecture (Ti-TMS320C62x) Source: [10]

2.1.2 Static Scheduling

The goal behind *VLIW* architecture was to speed up the execution of a program by running several operations in parallel, instead of running them in sequence, like modern Out-of-Order Execution (*OoOE*) architectures currently do. The main difference between *VLIW*

and *OoOE* architectures is how operation parallelization is made: while *OoOE* architectures use *dynamic scheduling*, *VLIW* architectures use *static scheduling*.

The core difference between these two operation scheduling techniques is the component that drives the scheduling process and in what moment it is made. Dynamic scheduling is done by the architecture using hardware, it's done at run-time and is always binary transparent; static scheduling is done at compile-time by a compiler/scheduler framework and is coded in the execution binary.

Because *VLIW* architectures use static scheduling, when compared with *OoOE* architectures, they do not need as much control hardware or spend power at run-time parallelizing operations. On the other hand, they require a more complex compiler, capable of targeting the parallelism of the *VLIW* architecture.

2.1.3 VLIW applications

VLIW architectures do not have only advantages. *VLIW* architectures require a more complex compiler framework due to the need for static compile-time scheduling. *VLIW* performance is directly linked to the amount of *ILP* a compiler can get from the program because execution units can only execute simultaneously operations that can be executed in parallel. If a program is composed only of non-parallelizable operations, the compiler must issue *NOP* operations to most execution units, reducing the execution performance of the processor.

To issue a control flow operation, such as a branch, compilers may not be able to issue operations to execution units without knowing the branch outcome. When this happens, conditional control flow operations can get as expensive as the number of execution units the architecture has.

Because of these disadvantages, *VLIW* architectures are not usually seen as a suitable alternative for the general purpose computing, although Intel did so with its Itanium architecture. *VLIW* architectures can however provide a powerful alternative for computation hungry kernels if properly configured and designed to target specific applications.

2.2 ICG technology

ICG's main role inside Intel is to provide complete imaging solutions for consumer Systems on Chip (*SoC*). Because image processing algorithms are easily parallelized in both data and instruction-level parallelism, the solution currently employed in *ICG*'s imaging solutions is based on multi-processor *VLIW* architectures.

The development of new processor architectures is a process that usually requires a considerable amount of effort and time-to-market. To counter this, *ICG* adopted an automated processor development flow, retargetable to a wide range of clustered *VLIW*, which reduces time-to-market, engineering effort and allows more efficient design space exploration.

ICG's tool flow is built around a *VLIW* processor template, the Processor Architecture Template (*PAT*), and contains tools capable of driving both Hardware and Software for processors designed according to it.

2.2.1 Processor Architecture Template

Despite some limitations, the template allows the customization of a practically unlimited set of parameters, supporting the development of a very wide set of *VLIW* processors.

Figure 2.2 shows a graphical representation of the *PAT*. The main components are the Register Files, the Issue Slots, the Function Units and the connection networks.

The role of the Register Files is to store data. In the *PAT*, there can be any number of Register Files with different configurations of sizes and number of ports. As Figure 2.2 suggests, by having multiple Register Files the template was designed to allow the development of clustered *VLIW* architectures but can also be used with single-register file architectures.

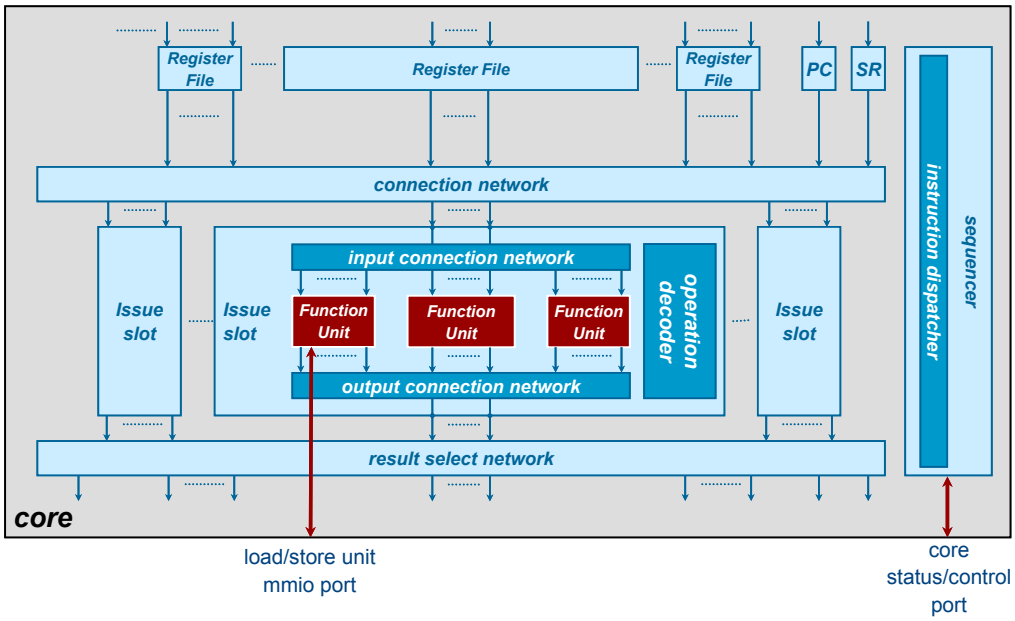


Figure 2.2: The Processor Architecture Template Source: [1]

Function Units have a finite number of input and output ports and define a finite set of customizable *Operations*. The selection of *Operations* to go inside a *Function Unit* usually targets a specific application, such that Function Units can later be chosen based on their purpose. Good examples of Function Units are *Arithmetic Unit* (with operations add, sub, add_i, sub_i, etc), the logic unit (with bit-wise operations like or, xor, and, not) or the branch unit (with jump, jump_if, jump_not_if). Function Units are used to build Issue Slots.

To customize an *Operation*, the developer can define its number of inputs and outputs (as long as the Function Unit has a suitable number of ports), its semantics and even its multi-cycle behaviour. An *Operation* can be either single-cycle or multi-cycle. If it is a multi-cycle operation, the developer can specify the exact cycles when the inputs should be fetched and when the results will be set in the Function Unit output port.

Issue Slots are the equivalent of *Execution Units* presented in Section 2.1. They have a set of customizable input and output ports and Function Units. To build an Issue Slot, the developer describes its input and output ports, its Function Units and the Issue Slot's input and output connection networks. The set of Operations an Issue Slot can execute is chosen

based on the Function Units used to build it.

The *input* and *output connection networks* are both fixed datapaths. They connect the input ports of the *Issue Slot* to the inputs of the *Function Units* and the output ports of *Function Units* to the outputs of the *Issue Slot*, respectively.

The *connection network* connects the output of the *Register Files* to inputs of the *Issue Slot* and is also a fixed datapath.

The last network, the *result select network*, is the only dynamic part of the template's datapath. This network is composed by several *Buses* and every cycle each Bus can transport data from one *Issue Slot* output to one or more *Register File*'s inputs.

At design time, the developer can configure multiplexers in the input of the *Buses* and at the input of *Register Files*. Thus, every cycle, the *Bus* can select one of several *Issue Slot* output ports and the Register File port can select one of several *Buses*. The port select value for such multiplexers is given in the *instruction word*.

2.2.2 The Tool Flow

As mentioned in the previous section, a complete tool-flow exists to work with the *Processor Architecture Template*. All the tools are driven by the same Machine Description file. This describes a particular instance of the template.

Figure 1.1 (page 2) shows a representation of *ICG*'s main development flow. The flow has two user inputs: the *Platform Design* and the *Application Design*. The *Platform Design* is specified through *TIM*, a processor description file, and *HSD*, a system description file. *Application Design* is specified through *ANSI-C* source files.

The *TIM* compiler generates a processor Machine Description from the *TIM* files (Figure 1.1's *P1*, *P2* and *P3*) and the *HSD* input gives information about the configuration of the system. The *HSD* compiler builds the system by connecting the individually compiled processors with a *System Bus* and a *System Memory* (not shown) to build a platform. GenSys can then generate synthesizable *RTL*.

The *Application Design* needs to be mapped to the System, first by parallelizing the application and second by targeting the system cores. *HiveCC*, *ICG*'s compiler, takes the mapped *ANSI-C* code and compiles it to the *VLIW* processors. A System Simulation framework uses the compiled platform and the compiled source code to generate simulation executables. The execution of the simulation executables generates performance metrics on the System and Application.

Although every tool is in one way or another connected to another tool, the tool set can be separated into two main flows: the hardware and the software flow. The hardware flow takes the machine description and pre-defined IP blocks and generates synthesizable *RTL* code. The software flow has as inputs the machine description and an *ANSI-C* source code and generates both simulation models to run on the *host* and machine binaries to run on the *target* architecture.

The software flow can be divided into another two flows, the compiler flow and the simulator flow. The compiler, as seen previously in Figure 1.2 (page 3), starts by mapping all

the operations of a compiling source code into the target machine using a C *front end* and an operation selection algorithm.

Once the source code is fully mapped to the processor's specific operations, the compiler calls the *scheduler* to schedule the code and exploit the best it can from the program's *Instruction Level Parallelism*.

Like shown previously in Figure 1.3 (page 3), the simulation flow takes the scheduled code as an input. Together with the operation semantics description given in the machine description, the scheduled code is transformed into a *C* simulation model by means of a *Compiled Simulator* generator. The generated simulation model is finally compiled and executed to obtain the simulation results in form of execution timing and tracing.

ICG simulation framework uses an in-house simulation scheduler and API to simulate multi-core systems. The compiled simulation models for the *VLIW* cores are linked to the simulation scheduler together with device models (such as system memory, DDR) to provide a system simulation.

2.3 SystemC

> "*As the electronics industry builds more complex systems involving large numbers of components including software, there is an increasing need for a modeling language that can manage the complexity and size of these systems. SystemC provides a mechanism for managing this complexity with its facility for modeling hardware and software together at multiple levels of abstraction. This capability is not available in traditional hardware description languages.*"
>
> (IEEE 1666-2011 SystemC Standard [2])

In its most crude definition, SystemC is nothing more than a C++ library. However, such library gives support for a complete simulation framework.

SystemC C++ classes include *Modules*, *Ports*, *Exports*, *Channels*, *Processes*, *Interfaces* and *Events*. The importance of all these classes is that they provide the support for a developer to divide his/her system into separate, self-contained modules (as a child of the *Module* class), the support to model the communication between them (using *Ports*, *Exports*, *Interfaces* and *Channels*, depending on the level of abstraction desired) and the support for Simulation scheduling through *Processes* and *Events*.

SystemC also provides support for HDL-like variables, like *N-bit sized* integers or N-sized *logic arrays*, and logical and arithmetic operations to operate upon them. The HDL-like data types and operations allow a programmer to specify the same kind of logic operations he would be able to do in any HDL language.

With all these features combined, a developer can model a full system according to the specific level of abstraction he needs. He/she can model part of a system using cycle-accurate simulation (original *SystemC* language) while modeling the rest as time-decoupled (*SystemC TLM*). Both *SystemC* and *SystemC TLM* are part of the IEEE 1666-2011 standard.

Furthermore, because *SystemC* is built on top of *C++*, the developer can model most of his devices either using plain *C++* code, by modelling the hardware details, or by using a mixture of both, as mentioned in the previous quote from the *SystemC* Standard.

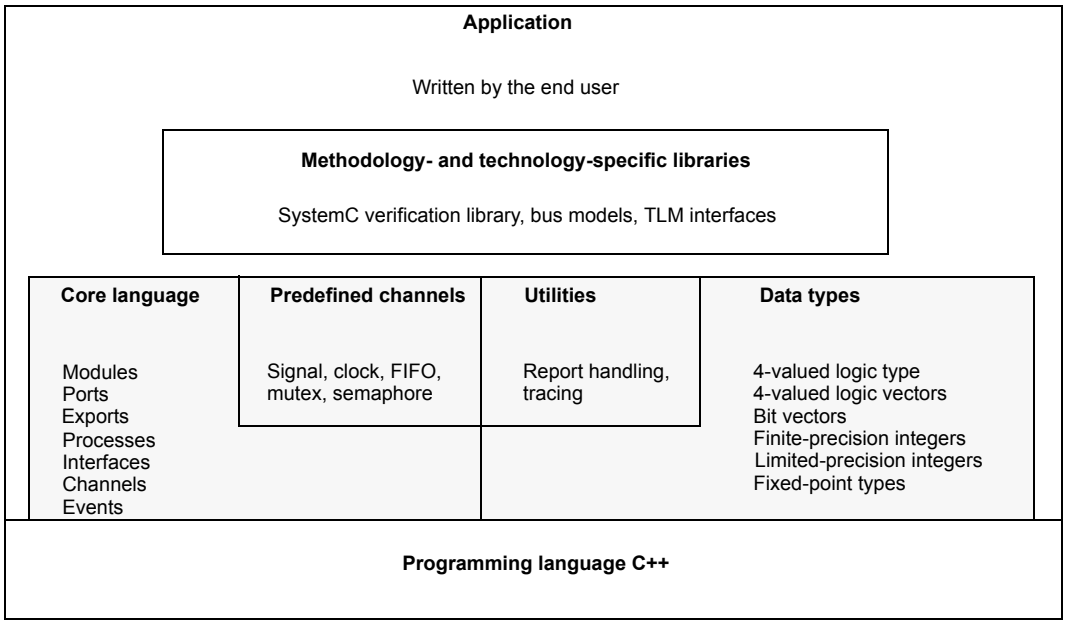


Figure 2.3: SystemC language architecture. Source: [2]

Figure 2.3 is a representation of a *SystemC* application architecture. A *SystemC* application is built using *C++* as the base programming language and the *SystemC* class libraries (shaded area), which can be divided into 4 categories: Core language, Data Types, Predefined Channels and Utilities.

The user can also use other libraries, like C++ standard libraries or methodology-specific libraries. These are not necessarily part of the IEEE SystemC Standard but can be used in a *SystemC* application. These libraries are represented by the floating box on top of the shaded area. Examples of such libraries are SystemC AMS (Analog and Mixed Signal), SystemC SCV (SystemC Verification) and SystemC CCI (Configuration, Control and Inspection), which are being developed and maintained by the Accelera Systems Initiative[11].

2.4 Random Program Verification

One of the most important phases in modern digital circuit design is verification. Due to the high complexity of current digital designs, it is practically impossible for any hardware developer not to have a bug or forget a corner case while developing *RTL* code.

Verification is the process by another engineer (or automated tool) verifies the behaviour of the production *RTL* against its design specifications, to search for inconsistencies between them. The earlier these inconsistencies are found and fixed, the less expensive it is to fix them.

One verification technique usually employed in processor verification ([7]) is called Random Program Verification (*RPV*). In few words, *RPV* aims to verify the functionality of a processor implementation by comparing execution results of a randomly generated program running in the *RTL* code against the results of the same program running in an of the same architecture.

By using this technique, the verification engineer does not have to develop programs by

hand in an attempt to cover the best of a processors' *ISA*. With *RPV*, an automated tool can generate a statistically significant amount of random instructions in an attempt to cover a statistically acceptable amount of the processors' functionality (due to size and complexity of modern computer architectures is infeasible to thoroughly test the complete *Instruction Set Architecture* of a processor).

2.5 Conclusion

In this chapter, some background on *VLIW* cores, *ICG* technology and the *SystemC* language was presented. This background is needed for this work as it presents a new SystemC retargetable simulation framework for *ICG*'s *VLIW* architectures.

Some background on Random Program Verification was also presented because it is one of the motivations for this work.

Chapter 3

Related Work

3.1 Retargetable Processor Development Kits

Besides ICG technology, there are other companies using the same approach towards Application Specific Instruction-set Processor (*ASIP*) development. A very good example of this is *Tensilica*, bought recently by *Cadence* and now called *Cadence's Tensilica-IP* [12][13].

Cadence's *Xtensa Processor Developer's Toolkit* provides, like ICG's tool kit, a complete set of tools to allow quick design and design space exploration of processor architectures, including synthesizable *RTL* generation for the same architectures. In Figure 3.1 we can see *Xtensa*'s tool flow.

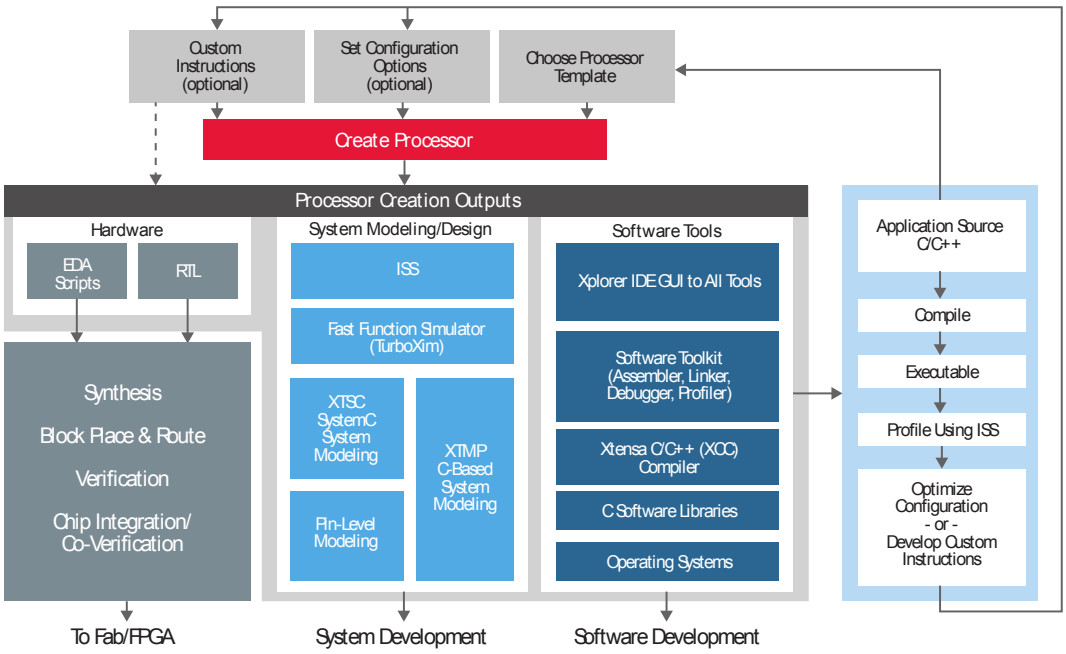


Figure 3.1: Tensilica Processor Development Toolkit's Flow Source:[3] [4]

One major difference between *Tensilica Xtensa* and *ICG*'s *PAT* has to do with the fundamental base architecture. While the first focus in maintaining a fixed RISC-based architecture which then can be extended by the developer to fulfil his particular needs (up to a maximum

of a 3-Issue Slot *VLIW*), ICG is based on a much more flexible *VLIW* based architecture with an unlimited number of Registers and Issue Slots.

The reason for *Tensilica Xtensa* to maintain the base core and a small architecture has to do with easily maintaining code quality by sacrificing some flexibility[14]. *ICG*'s approach relies on an in-house, very aggressive, optimizing compiler to do that job.

3.2 VLIW Simulators

On *VLIW*-specific simulation, [15] did some work on a simulator for the previously shown Texas Instruments TMS320C62x (Figure 2.1 in page 8). The approach was to create a separate simulation flow for the execution of operations and another for the control of the timing of the pipeline. In this approach, cycle-accuracy was also a major concern to ensure simulation correctness, but to accelerate simulation time individual execution units were abstracted as the author demonstrated that such level of detail was not need to be modelled.

In [16], the author introduces a static binary translation flow to execute cross-compiled binaries in a native simulation context. This is a compiled simulation approach and starts by interpreting the target binary, generating an internal representation of the program (using LLVM framework) and then using the internal representation to generate native code for the machine where the developer is currently working on.

3.3 Fast Instruction Set Simulators

On the matter of fast instruction set simulation, one can consider two main approaches: the compiled simulation approach and the binary translate approach. In [6], another approach of statically translated binaries like [16] is presented, but for more generic architectures.

Besides these two approaches, there is also an effort to speed-up instead Interpretive Instruction Set Simulators. Some techniques for speeding up *IISS* are given in [17], with special attention for the decoding cache, used in this work as pre-decoded lookup tables.

3.4 Retargetable Instruction Set Simulators

In [18] and [19], an approach based on LISA to describe the architecture is used. In both cases, and in order not to compromise simulation speed, a compiled simulator approach is used.

In [20] an approach to generate an Instruction Set Simulator (*ISS*) from a complete property suite is presented. In it, the developer describes the elements that constitute the status of the processor and the conditions to obtain the next state based on the current state and instruction word. In the end, a complete *Instruction Set Simulator* is generated for such property suite.

Chapter 4

Problem Definition

4.1 An Interpretive ISS for ICG's processor architectures

As presented in Section 1.2, *ICG*'s simulation solution (*Compiled Simulator* approach) provides a fast simulation solution but does not cover the full extent of *ICG*'s *VLIW*'s *Instruction Set Architecture*, particularly its binary interface.

An *IISS* provides such coverage as it executes the same binaries as the targets it simulates.

4.1.1 Binary Validation

Because of how the *Compiled Simulator* is generated (refer to Section 1.2 in page 2 and Figure 1.4 in page 4), there is a current lack of coverage on validating the output of the full compiler flow. The *Compiled Simulator* uses the output of the scheduler, which is a code *intermediate representation* and not the target executable binaries. Because there are some architecture-specific details implemented by the assembler, in particular the routing of data through the datapath, there is room for error in this last compilation step.

This does not mean that the only way to validate machine binaries is through an *IISS*. Currently this is being done by running a *RTL* simulation of the target architecture. Although very accurate, *RTL* simulations are complex and slow because they run at Register Transfer Level. *RTL* simulations are mainly used for purposes of testing and validating hardware code and not program binaries. This means that verifying binaries through these simulations can become cumbersome and, to some degree, painful for the developer when compared to running binaries using an *IISS*.

4.1.2 Random Program Verification

The second motive to develop an *IISS* is enabling full Random Program Verification (*RPV*) flexibility within *ICG* tool kit. As presented in Section 2.4, *RPV* is a known process to validate the functionality of an implementation of a given architecture.

Because *ICG* tool flow currently does not have an *IISS*, engineers are validating the functionality of the architectures using the *Compiled Simulator*. The *Compiled Simulator* approach does not support simulation of target binaries, only *internal representations*. The higher-level *internal representation*, generated by the *RPV* tool, needs to be encoded by the

assembler to target the processor. This can leave room for untested corner cases, as the assembler does some processing of the internal representation.

Besides, the current *RPV* framework needs to generate all test programs before starting the verification process. This is a consequence of the compiled simulator framework that requires the *intermediate representation* files to generate simulation models.

With an *IISS*, the framework would be able to launch the *RTL* simulator and *Instruction Set Simulator* simultaneously and inject both with the same just-in-time generated instructions. This approach has the advantage of only requiring to be launched once. With the compiled simulator approach, *RPV* needs to be reset and re-started at every iteration.

4.1.3 Self-modifiable Code

The third motive is to enable simulation of self-modifiable code. *Boot-loaders* are a powerful type of application used to enable reprogramming embedded processors without specialized hardware. *Boot-loaders* are usually very small programs stored at the beginning of the program memory, which can re-write parts or all the program memory, accordingly to an external input.

When a processor is released from a reset, the *boot-loader* checks for a re-programming condition. If such condition is detected, the *boot-loader* loads a new program into the processor, given by the external input; if not, it starts executing the remaining program in the program memory.

Because current simulation techniques rely on a static compiled simulation framework, modeling the full behaviour of a *boot-loader* is currently not possible. Also, current simulation techniques do not detect erroneous writes to the program memory as it is not used in the simulation.

4.1.4 Fixed Simulation Model

The last motive to develop a new *IISS* is to have a fixed simulation model. The processor development flow from the high-level Machine Description to the final Silicon implementation is not fully automated. The generated *RTL* code can be changed and are parts of the Silicon process which are done by an engineer. Thus, there is room for manipulation and extension of processor features beyond the *PAT* capabilities.

Compiled Simulator models are unique for every program a software developer writes. Even if a single line is changed in the source code, the simulation model needs to be re-generated and re-compiled.

This represents a challenge in reproducing extensions done in non-automated development stages to the simulation models because the *Compiled Simulator* generator needs to be modified. By having a generated fixed simulation model *Instruction Set Simulator*, the user can modify the model source code without modifying the tool.

Having a fixed simulation model also enables additional *backwards compatibility* checks. As the tool set evolves into newer versions, it is important to make sure old architectures are still supported by the compiler.

A copy of the simulation model for an architecture can be kept as a reference and newer versions of the compiler can be tested with the old simulation model to make sure the old architecture is still fully supported.

Also, with a fixed simulation model, the developer does not need to wait for two compilation cycles to run a simulation of his code. Currently, first the tool flow needs to compile code for the target processor and then compile the *intermediate representation* to a simulation model.

With a fixed simulation model, the developer only waits for the compilation of the target code. Simulation can be launched as soon as the first compilation step finishes.

This is only relevant for the development of small kernels as, with bigger kernels the time spent in re-compiling the simulation model is practically insignificant compared to the overall simulation time.

4.2 Requirements

The following section presents the requirements for the proposed solution. These requirements provide a basis to address the problems presented in Section 4.1.

4.2.1 Compatibility with framework / toolflow

Developing a new *IISS* has the goal of complementing the tool flow without disrupting any of its features. To achieve this, the new *IISS* should be able to handle any processor architecture described according to the *PAT* and use the same Machine Description file currently used by the tool set.

4.2.2 Binary transparency

Binary transparency means that the programmer should see no difference, at the Instruction Set Architecture (*ISA*) level, between running a binary on a target architecture or on the simulator.

It does not mean that the simulator should be another *RTL* implementation; it means that the *Instruction Set Simulator* should mimic, with precision, the external behaviour of the processor's *ISA*.

Binary transparency is a requirement by definition of any *Instruction Set Simulator* but specially in this case, as one of the main motivations is to enable easier and faster binary verification.

4.2.3 Modularity

The solution should try to follow the guidelines of modular programming to enable better maintainability and easier manipulation of the simulation models.

The concept of modular programming is to separate the full functionality of any software into smaller blocks that execute a particular, smaller part of the whole functionality.

In this way is easier to debug, maintain and extend the simulation model both during its development and along its life time.

4.2.4 Human Readability

Human-readability of the *Instruction Set Simulator* code is necessary to allow any engineer to address the problem presented in Section 4.1.4.

If it is desired to manually extend the simulation model beyond what the *PAT* currently allows, no matter if the developer intends only to reduce simulation times or extend the processor model, it is crucial that he is able to understand what is written in the simulator source code.

Besides enabling extensions to the simulation model, having a human-readable source code also eases maintainability, specially when a developer needs to debug or modify code that its not his own.

4.2.5 Fixed Simulation Models

This requirement directly addresses the problem presented in Section 4.1.4.

A fixed simulation model for each architectures creates room for easier development of extensions or manual optimizations. It also allows maintaining copies of old architectures simulation models to ensure backwards compatibility of newer versions of the compiler.

To achieve that, the solution should be able to provide a fixed simulation model for each architecture, independent from the target program.

4.2.6 Integration in Simulation Frameworks

The last requirement is the integration of the *Instruction Set Simulator* model into other simulation frameworks.

In a multi-core / multi-device environment, support for the simulation of different models at the same time is needed. Besides, current compiled simulators interact only with *ICG*'s internal simulation models. This does not allow easy integration into other industry simulation frameworks.

The solution should be able to be integrated into a well-known industry system simulation framework, like SystemC.

4.3 Challenges

Because the flow is retargetable and because of the flexibility provided in the *PAT*, creating an *IISS* that can simulate any of these *VLIW* architecture will present challenges that will be addressed in the following sections.

4.3.1 Retargetable Framework

Addressing a Retargetable Framework is a challenge. When targeting a single architecture, access to its entire specification is available to the developer during the construction of the simulator. Besides, the simulator does not need to support any other architecture.

In a retargetable framework, specially one based on an Architecture Description Language, the full targetable set of architectures is practically infinite.

This represents a challenge as the new simulation framework must be able to target any configuration (according to the template) without loss of accuracy. Furthermore, because of

such large design space, it is also impossible to validate the simulator against every possible configuration. This might leave room for error if every feature is not thoroughly tested.

4.3.2 Operation Timing and Semantics

As mentioned briefly in section 2.2, the developer has the ability of customizing the semantics and timing of all operations of an architecture.

The challenge of operation semantics is on their level of description. The semantic of an operation can be defined with an HDL-level syntax, which allows the description of N-sized arithmetic and logical operations. An operation can also have more than two inputs and more than one output.

Operations in the *PAT* do not need to be executed in a single cycle and the developer has the power to create multi-cycle (pipelined) operations. This is an important aspect, specially for complex operations, like divisions, which require several cycles in hardware to be executed.

These multi-cycle operations can even allow the fetch of input data to be done in different cycles along its execution, further increasing their level of flexibility.

Because the simulation results need to be bit-true, this two factors make reproducing semantics and operation behaviour challenges on developing an *Instruction Set Simulator*.

4.3.3 Simulation Performance

It is a known fact in the academia that, put in the same conditions, *Interpretative Instruction Set Simulator* are significantly slower in execution then *compiled simulators* [6]. However, this does not mean that the proposed solution should not try to be as fast and efficient as possible.

Simulation Performance as a requirement means that although the *IISS* is expected to be slower compared to the current *Compiled Simulator*, the difference should be acceptable from the developer's point of view.

Chapter 5

Proposed Solution

There were two possible approaches to a new retargetable *Instruction Set Simulator*:

Run-time configurable IISS This first approach would be a self-retargetable simulator, where at run-time the Machine Description would be read to re-configure the *IISS* to the last settings.

This approach would not fulfil some of the requirements presented, as having a fixed simulation source code or performance, due to considerable run-time configurations it would require. Besides, it would be complex and hard to fully implement.

Generated IISS This second approach would be a tool capable of generating the source code for an *IISS* of an architecture read from the Machine Description. The generated source would be compiled to an *IISS* executable, specific for every architecture, but able of being integrated in current simulation flows.

With the second approach, besides powerful compilation-time optimizations that modern compilers are capable of performing, the fixed simulation model requirement is fulfilled (the generated file would not need to be regenerated again for the same architecture).

The technology chosen to develop the *Instruction Set Simulator* was SystemC because it is one of the de-facto simulation environments, used both in the academia and in the industry. Also, because it has native support for HDL-typed variables which were used to implement the fully customizable operation semantics.

This chapter will present a proposal for a generated *IISS* and its structure and behaviour. A section on the generation tool is also presented here.

5.1 IISS Generated Model

5.1.1 Simulation Main Flow

The role of an *Instruction Set Simulator* is to mimic the behaviour of an *ISA*. An Interpretative Instruction Set Simulator (*IISS*) achieves this by keeping an accurate internal representation of a processor status and by interpreting the instruction words of a program.

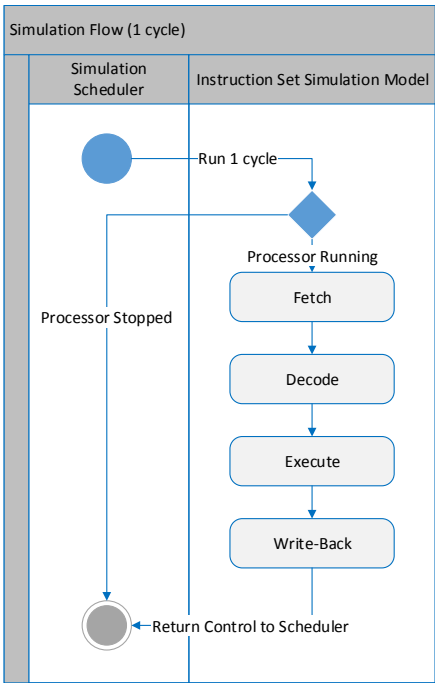


Figure 5.1: Instruction Set Simulation activity diagram of one cycle's execution

The activity diagram of instruction word interpretation can be visualized in Figure 5.1. Figure 5.1 distinguishes 4 phases. The first phase, *fetch*, corresponds to the load of a new instruction word from the program memory. *Decode* corresponds to the decoding of the loaded instruction, this is the extraction of the execution information given to the processor in the instruction word.

Execution corresponds to the dispatch of operations given in the instruction word and *write-back* corresponds to the store of execution's results into the *register files*.

Source Code Structure

To fulfil the requirement for a modular solution, the proposed approach divides the processor simulation model into self-contained sub-modules. The choice of modules took into consideration the *PAT* classifications as the role of the different blocks can be classified as part of *fetch*, *decode*, *execute* or *write-back* phases. The proposed separation of modules is shown by means of a UML class diagram in Figure 5.2 and how those modules fit into the simulation flow is shown in the activity diagram in Figure 5.3.

In Figure 5.2, we can see the *Core* class as an hierarchical top. The *Core* Class is composed by data *Memories*, a *Program Memory*, *Issue Slots*, *Register Files* and *Buses*. All these elements are instances of separate classes. The *Core* class also implements the decoding methods used in the decoding phase.

The contents of the *Issue Slot* and *Register File* are also shown in Figure 5.2. The only attributes of the Issue Slot are the Pipeline Queue, called *PipelineOperations* and the results of the latest cycle execution phase, called *SlotOutputs*. The *Issue Slot* class also implements

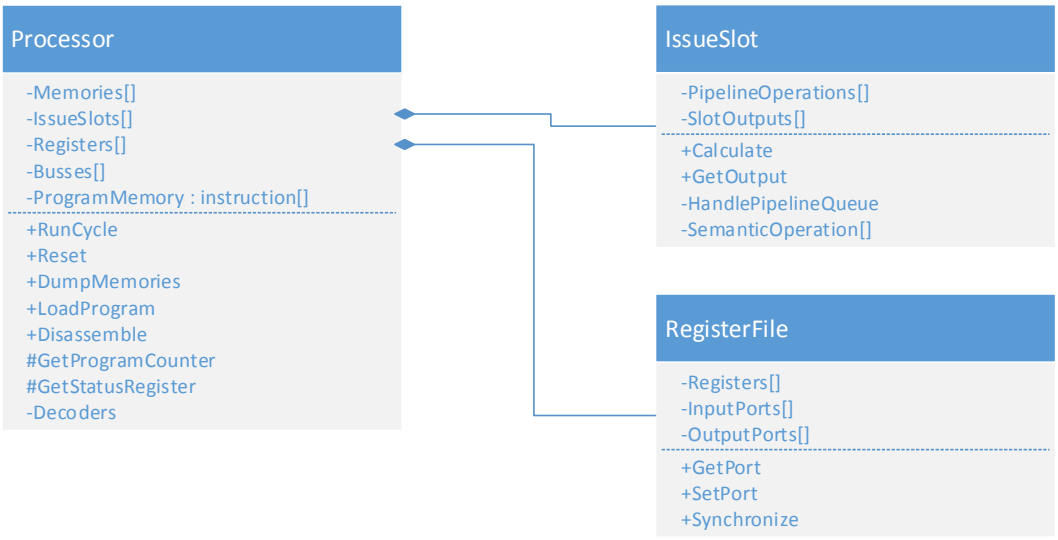


Figure 5.2: Core Class Diagram

internally the semantics of operations it can execute.

The Register Files need the value of the Registers, the value of input ports and output ports. The goal of keeping input and output ports as separate attributes is reproducing Write and Read Latency during simulation.

Source Code Activity

Figure 5.3 shows the phases of Figure 5.1 mapped to different elements of the *Core* class. The execution of one cycle in the *IISS* solution starts by verifying the Status Register of the processor. If the processor is stopped, control is returned to the Simulation Scheduler, otherwise execution of a new instruction starts.

First a new instruction is fetched from the program memory and decoded (fetch and decode phases), at the top class level. Then, using the decoded register indexes, the execution phase starts by fetching the input data from the Register Files. This data, together with the decoded opcodes, is dispatched to the Issue Slots using the Calculate and Pipeline Handle calls, ending the execution phase.

The write-back phase starts by fetching the results from the Issue Slot and, after taking into consideration the datapath, ends by writing the results into the register files and returning control to the scheduler.

Implementation

To allow easy manipulation and understanding of the simulation flow, the generated code is clearly separated in the four phases previously shown in Figure 5.1 by means of comments.

This separation can be seen clearly in the pseudo-code shown in Listing 5.1, which is an extraction of what the generated code should look like. The names of the components

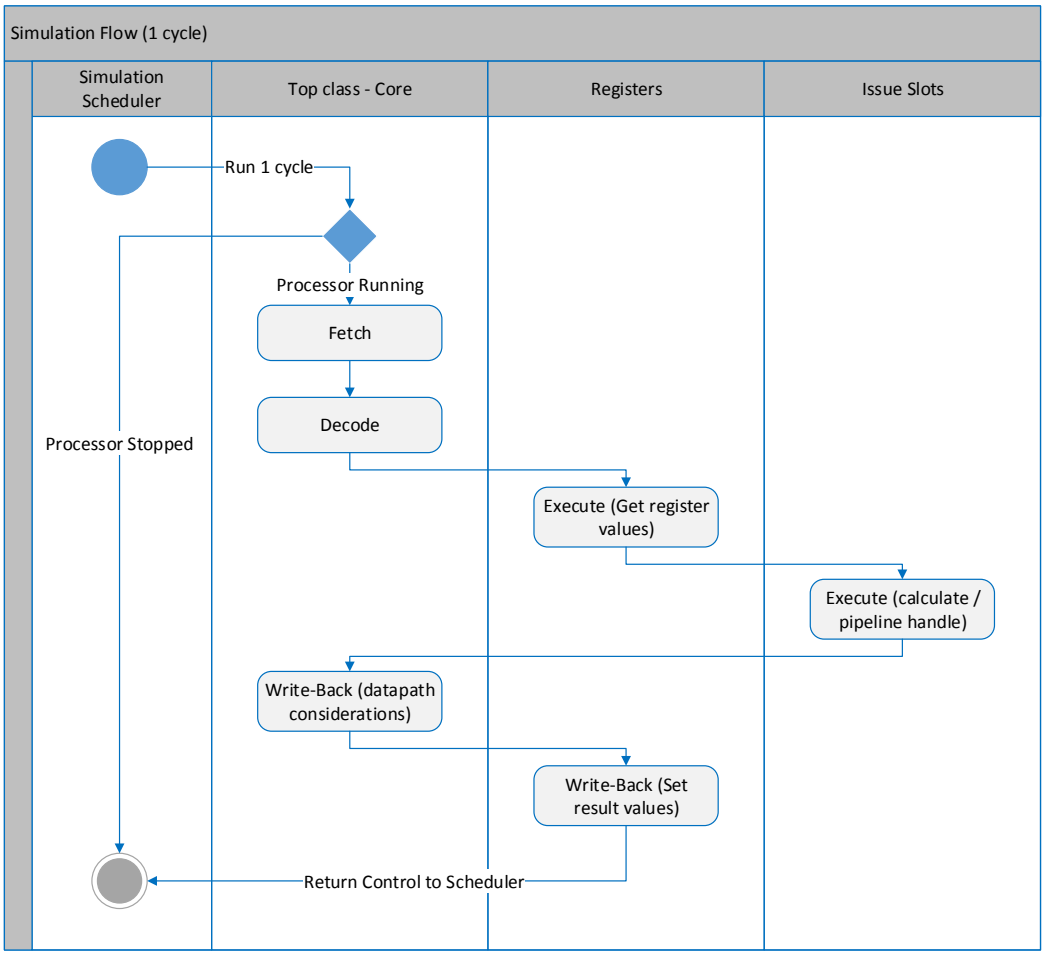


Figure 5.3: UML activity diagram for the different elements of the simulation template

---

of the *PAT*, here named after their types, are replaced by the names given in the Machine Description file.

Listing 5.1 also shows a more detailed implementation of the flow explained earlier. The flow starts by the fetch of a new instruction word. Then, the instruction word is decoded to temporary variables, named after their use (per example, slot0_opcode is the opcode for slot0, and so on).

The execution phase starts by synchronizing the Register Files. This step is important to reproduce the Read / Write Latencies that a register might have. The fact that Register File read indexes are used in the *Synchronize* method and not on the *Get* is to avoid user confusion, as with Read Latencies the result of a *Get* could return a register of different index from the one used in the method call.

After the Register File Synchronize, the outputs of the registers are fetched to temporary variables. These values are then used together with the issue slot opcode and immediate to call *Calculate*. The *Calculate* method handles a new Issue Slot operation and advances its pipeline queue by one cycle.

In the Write-Back phase, the output values of the Issue Slot are fetched to the Write-back Buses. The Write-Back has two separate stages due to two levels of multiplexing in the

datapath. In the first level, the Buses select a specific Slot output to fetch data. In the second level, the Register File's ports select one of the Buses. After the Write-Back phase, control is returned to the scheduler.

5.1.2 Operation Execution - Issue Slot Flow

Executing operations while maintaining the same behaviour as the hardware is one of the challenges that this work needs to address. The proposed solution implements the entire operation semantics inside the Issue Slot Class. This is done by creating a custom *Issue Slot* Class for every Issue Slot in the Machine Description file.

The structure of the Issue Slot was already shown in Figure 5.2. Listing 5.2 shows in more detail the flow of the *Calculate* method.

In the *Calculate* method *switch*, a *case* for every *opcode* exists. If the *case* results in a single-cycle operation, the value is immediately calculated and the result is set to one of the output ports of the Issue Slot. In case it is a pipelined (multi-cycle) operation, then a new pipeline operation is added to the pipeline queue.

After the main *switch*, the *HandlePipeline* method is called. Here, an iteration is done over all the operations currently in the pipeline queue, advancing all one stage each. The reason to do so has to do with the flexibility of multi-cycle operation timing (presented as one of the challenges in Section 4.3.2).

Because a multi-cycle operation may fetch data, produce a result or both at any cycle (during its execution), there is a need to call every pipelined operation in the pipeline queue, every cycle, to make sure it's properly handled. This way data that should be fetched or calculated does not get accidentally lost.

This could be optimized in the sense that an event system could be implemented to only call at the right cycles, but because maintaining easy readability is desired and a considerable amount of multi-cycle operations may exist in any architecture, this simpler approach was preferred.

The semantics for every result are maintained in a separate function with the same name as the operation itself (with prefix *semantic_*). This way complicated semantics can be kept separately from the *switch* environment, maintaining a cleaner solution than writing semantics in the middle of said *switch*. If the semantics are small enough, the compiler should take care of in-lining them at compile time with aggressive optimizations.

5.1.3 Binary Transparency - Decoders

In the *PAT*, there are no different types of instructions. All the instructions share the same format and have fixed *bit-fields* for *opcodes*, *immediates*, register file *indexes* and *bus multiplexer indexes*. Because of this, all decoding *bit-fields* are totally independent from each other, making the implementation of decoding algorithms rather trivial.

The implementation of the decoding algorithms is, nevertheless, done using the modular approach by implementing each decoding bit-field as a separate function. In this way, if for any reason there is a need to change or replace the implementation for a particular decode, the developer can easily do it by changing the respective decoding function.

```
//FETCH
instruction = program_memory[program_counter];

//DECODE
register0_indexes = DecodeRegister0Index(instruction);
register1_indexes...

slot0_opcode = DecodeOpcodeSlot0(instruction);
slot0_immediate = DecodeImmediateSlot0(instruction);
slot1_...

bus0_multiplexer = DecodeBus0(instruction);
bus1_multiplexer...

register0_select0 = DecodeRegister0Select0(instruction);
register0_select1...

//EXECUTE
//Synchronize registers with new cycle
//Needed because there may be delays both in input and
//output of register files
register0->Synchronize(register_index0, ...);
register1...

//Fetch register files output values
register0_output0 = register0->Get(0);
register0_output1...
...

//Trigger Issue Slot calculation. In the calculate call
//Issue Slots also handle pipeline stages, so there
//is not a need for a separate call for that
issue_slot0->Calculate(slot0_opcode, slot0_immediate,
                       register_output0, ...);
issue_slot1...

//WRITE BACK
//First multiplexing level
switch(bus0_multiplexer) {
    case 0:
        bus0 = issue_slot0->GetOutput(0);
        break;
    case 1:
        bus0 = issue_slot1->GetOutput(0);
        break;
        ...
}
switch(bus1_multiplexer){...}

//Second multiplexing level
switch(register0_select0) {
    case 0:
        register0->Set(0, bus0);
        break;
    case 1:
        register0->Set(0, bus1);
        break;
        ...
}

switch(register0_select1) {...}
...
return;
```

Listing 5.1: Execute1Cycle

```
Slot::Calculate(opcode, immediate, register_output...)
{
    switch(opcode) {
        case NOP:
            break;
        case op_not_pipelined:
            semantic_operation(&slot_output, register_output...);
            break;
        case op_pipelined:
            start_pipeline(operation_pipelined);
            break;
        case....
        ...
        default:
            assert(0);
    }

    this->HandlePipeline(register_output...);
}

Slot::HandlePipeline(register_output...)
{
    for( pipeline_slots ) {
    //handle pipeline stage and go to next
        //current stage has an input -> fetch value
        //current stage has a result -> calculate result
}
```

Listing 5.2: Issue Slot Structure

5.2 The Tool

To meet the first requirement, the proposed solution is to generate the *IISS* directly from the existing Machine Description file. To do this, a new tool capable of parsing the current machine description language and from that generate SystemC source code for a processor simulation model is presented.

The high level design of this tool, shown in Figure 5.4, starts by reading and copying the Machine Description file into an internal representation created to support the same type of configurations as the *PAT*.

After having an internal representation of the target architecture, it triggers a simulation model builder (here called *SystemC* Builder) which will use the created internal representation to generate a *C++ / SystemC* simulation source code.

In the end, the generated *Instruction Set Simulator* is compiled using *GCC / G++* and linked against the SystemC Libraries to provide a simulation model executable.

To better understand the internal flow of the tool and its relationship with the *PAT*, a detailed explanation of the different generation steps is done in the next sections.

5.2.1 Overview

Like the generated model, the generation tool should be easily maintained in the future. Because of this, a modular solution was implemented for the tool itself and an Object-Oriented Programming approach was also used. The code of the tool was written in Python due to its strong support for string manipulation.

The tool, internally called by the codename *Sapphire*, is divided in three main components:

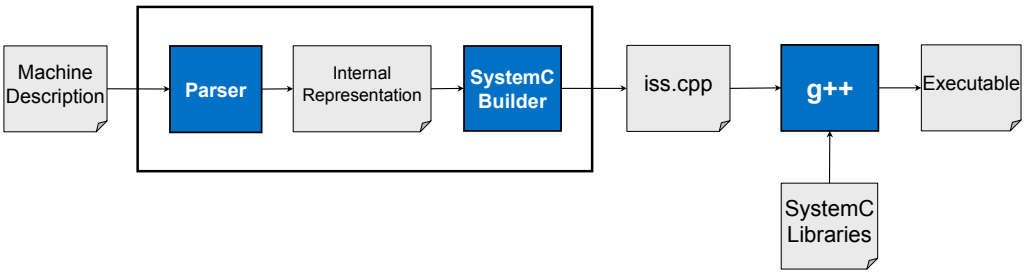


Figure 5.4: ISS generation (to merge into current flow)

The Core Description (a set of classes that mirror the *PAT* composition and that are used to build the internal representation), the parser (that builds the IR for every architecture) and the Builder, that uses the IR to build the proposed *Interpretative Instruction Set Simulator* Model.

5.2.2 Internal Representation - Core Description

The Core Description is a set of classes that represent all the elements that are part of the *PAT*. In Figure 5.5 (page 30) an UML class diagram of the Core Description package is presented. Due to the size and complexity of this package, a simplified version is presented.

As seen in the Figure, the main type of relationship between the different elements is composition. The reason to do so is closely related to the way the *PAT* instances are built. In the *PAT*, a Function Unit is a set of Input and Output ports and Operations and an Issue Slot is a set of Input and Output ports and Function Units, etc.

Using this approach, the Internal Representation can be easily built Top-Down from reading the Machine Description File. Also, because the structure can be easily traversed Top-Down, it is easy to search for and find any particular element.

Besides composition, there are also other kinds of relationship between different elements of the *PAT*. Association is used to model connections between different elements and inheritance, to avoid duplicating classes that could share most of their attributes.

The good side of using association to model connections is that it enables quick checking of the models' integrity and generation of a sane *Instruction Set Simulator* source code by the Builder.

5.2.3 Parsing / Building the Internal Representation

To construct the Internal Representation of the Machine Description of an architecture, the parsing algorithm uses three phases:

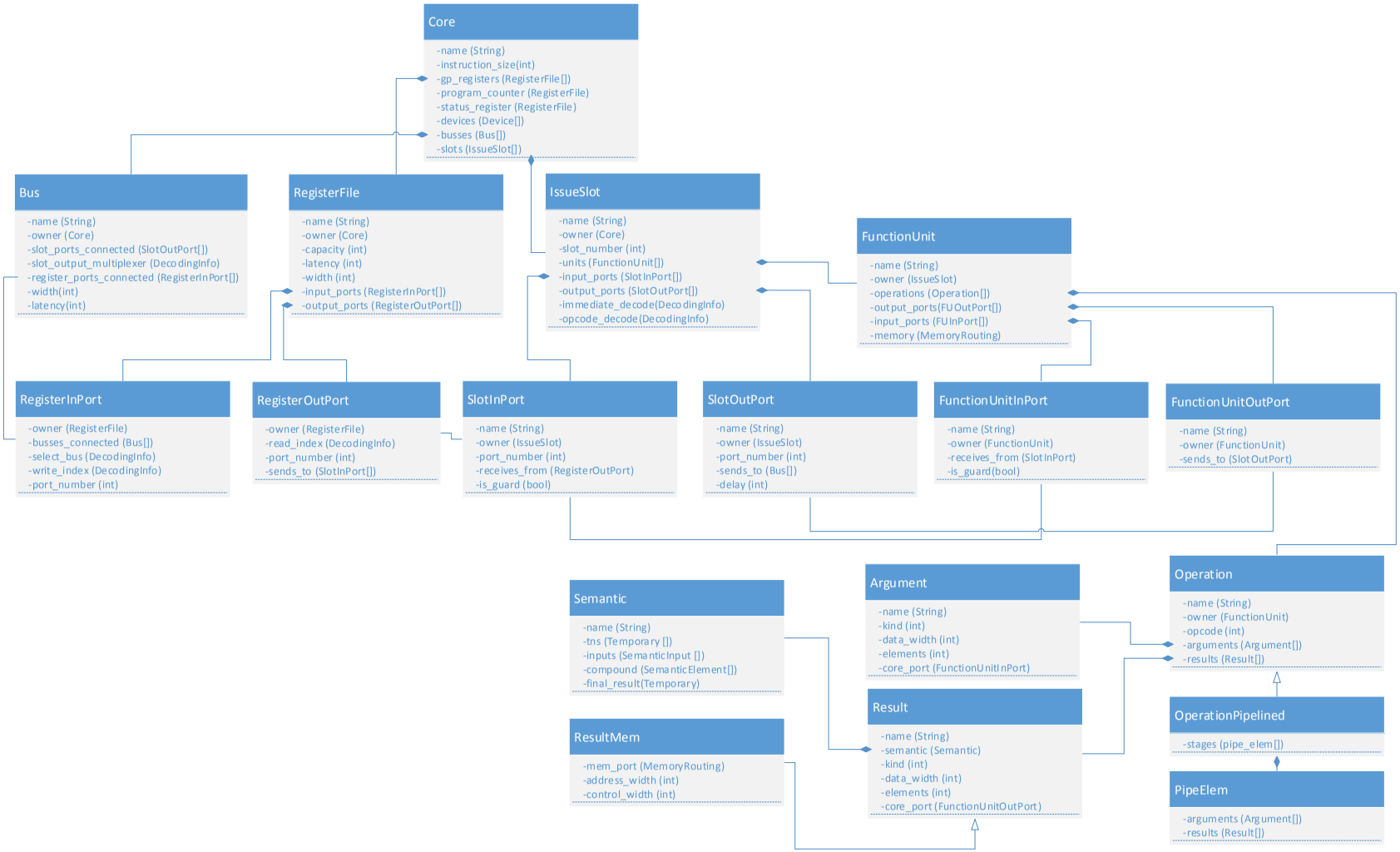


Figure 5.5: Internal Representation (simplified)

Creating Objects The first phase is the instantiation phase. Top-Down, the algorithm starts by creating a new *Core* then new *Register Files*, *Buses* and *Issue Slots*, then *Function Units* and so on.

In the end, the result is a skeleton of the Core Description, where all elements have been created but the only initialized attributes of every element are the *Name*, *Owner* and the reference to its sub-elements.

Parsing Individual attributes The second pass parses the individual attributes of every object. As an example, in the Core class, the instruction size is initialized (the remaining attributes are sub-elements, which have already been initialized in the previous phase) and in the Issue Slot, the *opcode* and *immediate*'s bit-fields.

In the end of this phase, all the attributes that are not a connection to another element have been initialized.

Connecting the Model The last phase is the connection phase. In this phase, for every connection, a search is done top-down for the names of the two elements of a connection. When a match is found, the connection is created between them and the algorithm starts again for the next connection, until all connections have been made.

In this phase, most of the model checking is done by verifying that a connection of the Machine Description input actually makes sense. If a connection described in the input file cannot be created, either because one of the target elements does not exist or because it is between two elements that should not be connected (violating the *PAT*), an error is generated and program execution stops.

By the end of this phase, the Internal Representation is complete and the tool can start building the Source Code for the Instruction Set Simulator.

5.2.4 Building the Simulator's Source Code

Having verified the Internal Representation, the program proceeds to build the simulation model. The main *Builder* behaviour is based on a *visitor pattern* because using this pattern leaves room to later expand the tool for other simulation frameworks, besides *SystemC*, and even to expand or change the *PAT* itself.

The *Builder* algorithm also has a total of 3 phases. In the first phase, two *strings* are created for the Core Class (.h and .cpp files) and another two for every Issue Slot. These *strings* are always initialized with a *skeleton* file, which defines a template for every Class source code. The reason to use the *skeleton*, instead of generating everything, is because there is a significant amount of code which is always the same, no matter the instance of the *PAT*.

The second phase is where the Visitor Pattern starts working. Starting by the Core Class, the visitor object is passed top-down along the Core Description. As it visits every element, the necessary simulation code is being generated and put inside the *skeleton* strings created in the previous phase. In this phase, the connections created in the Internal Representation are used. As the Visitor writes calls to Register Files or *Slot::Calculate*, it uses these associations to get the names of the objects directly from both ends of a connection.

The third and last phase of the building process is cleaning and writing output files. In this phase, an algorithm passes through the entire generated file to remove remaining (and

```
${core_header}

/*SystemC library includes*/
#include <systemc.h>

/*Other headers*/
${other_headers}

/* ... */
${decoders}

void
${core_name}::FetchDecodeExecute() {
    program_counter = get_program_counter();

    const instruction_type instruction = program_memory[program_counter];

    /* Decode indexes for registers */
${register_decode_indexes}

    /* Synchronize register input and output delays */
${register_synch}

    /* Fetch register outputs */
${register_connectivity}

    /* Issue slot calculate calls (decoding is called inline with the call) */
${issue_slot_execution}

    /* First Write-Back stage, fetch results from issue slot's outputs */
${bus_result_multiplexer}

    /**
     * Second Write-Back stage, write results into register files,
     * depending on current instruction bus multiplexer values
     */
${register_write_back}
}

/* ... */
```

Listing 5.3: Core Class Skeleton (simplified)

---

no longer needed) parts of the *skeleton* and also exceeding vertical white-spaces in order to have a clean output file. In the end, the Makefile to compile the source code is generated and the generated files are written to the output folder provided by the user.

5.3 Simulation Performance

As seen in later Section 6.4 and in Figure 6.5, the first implementations of this solution had a significant weak performance when compared to the existing simulation framework. Although this was expected from the beginning, performance is nevertheless one of the requirements of the solution, and that is the reason why optimizations were added in later stages of the development.

5.3.1 Pre-decoded lookup tables

As shown in Section 6.4, performance significantly degrades for architectures with instruction word sizes bigger than 64 bits. This is not by chance and the SystemC manual warns developers about handling data with sizes bigger than 64 bits as it significantly degrades simulation performance when compared to regular data variables (data variables bigger than 64 bits cannot fit in a regular processor register and need to be stored at all times in the memory, which is significantly slower).

Pre-decoded lookup tables were added later to the proposed solution because in every cycle a considerable amount of processing is done on this kind of variable (in particular, the decoding process). The idea behind these lookup tables is to diminish or remove the load on the decoders while executing programs that do not re-write themselves (self-writing code).

When the simulator starts a new simulation, the entire program is pre-decoded and only then executed in a faster fashion, as decoders simply access the lookup tables instead of decoding instructions repeatedly every cycle.

The proposed solution updates the lookup tables if any modification is done to the program memory, as seen in Figure 5.6, because binary transparency is a requirement and self-writing code (or an external re-write of the program memory) can happen.

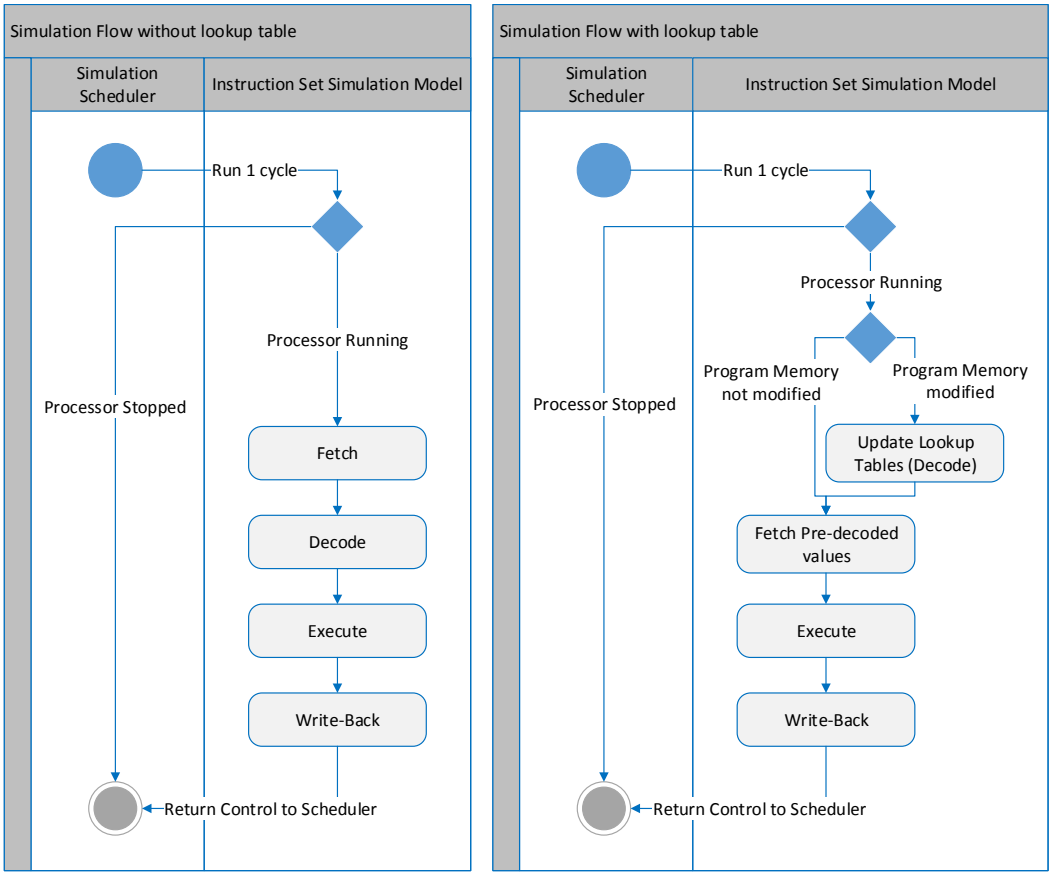


Figure 5.6: Lookup tables activity diagram

Simulation times are reduced significantly when compared to the previous on-the-fly de-

coding version because the need for constant decoding calls disappears. Because binary transparency is not compromised, this optimization is set by default in the final solution.

5.3.2 Decouple cycle-accuracy

While profiling some of the simulator binaries, specially those for small architectures, there was a considerable percentage of simulation time spent on the *SystemC* simulation scheduler.

An opportunity to create some kind of simulation time-decoupling appeared because the ability to validate compiler binaries don't require the simulation scheduling to maintain a cycle-accurate relationship between the processor model and remaining simulation models. However, because of how multi-cycle operations need to be executed, internal cycle accuracy was kept.

This optimization allows the creation of an internally cycle accurate model, but only returns control to the scheduler once at every N cycles. This optimization is shown later on the evaluation with N=32 (figure 6.7).

Chapter 6

Evaluation

6.1 Methodology

To evaluate the solution, several aspects were taken into account, from verifying that the solution provides correct simulation results to its run-time performance.

6.1.1 Validation

The first aspect to take into account when evaluating the solution is verifying it. There is no point in evaluating performance if the solution does not provide correct simulation results in the first place.

To evaluate this, comparison of simulation results was done between the current existing simulation platforms and the presented solution.

There are currently three important outputs of the simulation framework: number of cycles a kernel takes to execute, trace of its execution and final contents of the processor's memories and registers.

The criteria for acceptance in this matter was a complete match between the proposed solution and current existing compiled simulator framework.

6.1.2 Performance

As mentioned in the requirements, good performance metrics are also needed to not overburden developers using the proposed *Instruction Set Simulator* framework instead of the current compiled simulator. In any case, since the proposed solution is an interpretive *Instruction Set Simulator*, a worse performance is expected.

The metric used to measure performance is the simulation time (in seconds) of different benchmarks running on processors of different degrees of complexity. The exact same benchmarks were ran in the current compiled simulator framework and in the proposed *Instruction Set Simulator* framework, both compiled with the same versions of compilers and optimizations.

To compress the results into a single metric per core, geometric mean was applied to the simulation times of the different benchmarks.

6.1.3 Overhead

The overhead of re-generating the simulation models is also taken into account because any change in the architecture requires a re-generation of the proposed solution.

Overhead is the effort necessary to re-generate the *Instruction Set Simulator* after any change in the Machine Description.

The metric used to measure the overhead was the time (in seconds) necessary to both re-generate and re-compile the simulation binary from a clean directory.

6.2 Test Bench

To validate the solution a test bench was assembled. This test bench is a set of widely used benchmarks (PolyBench/C[5]) and a subset of current existing processors of *ICG*'s regression set.

6.2.1 Overview of PolyBench/C

Polybench/C is a set of C benchmarks used already as a reference in several academic papers [21][22]. It comprises 30 kernels with compile time parameters to adjust the size of kernel processing inputs.

Among the kernels provided by *Polybench/C* are several linear-algebra kernels and solvers, data-mining algorithms and image-processing.

The sub-set of *Polybench/C* benchmarks ported and used can be found in table 6.1. A sub-set is used because porting all the benchmarks to the tool flow would be cumbersome and unnecessary to the purpose of this document.

6.2.2 Core Set

To test the benchmarks with, a range of processors was selected from the current tool set regression kit. The criteria of choosing cores had to do with trying to cover as much as possible the *PAT* while disclosing the least amount of proprietary information as possible. The complete list of cores can be seen in Table 6.2.

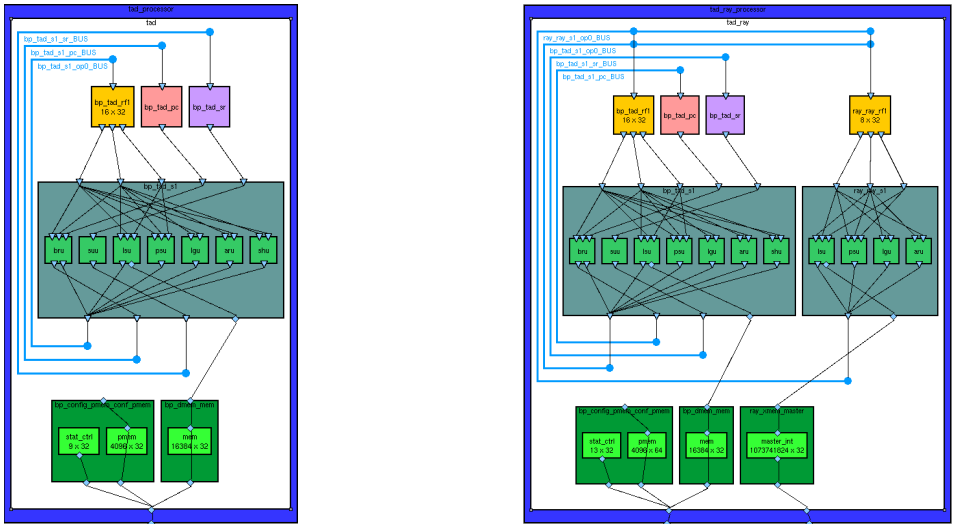
Tad / Tad Ray Family

Tad is the smallest and simplest core of the entire regression set. As it is a single *issue slot* processor, its *ISA* is comparable to a Reduced Instruction Set Computer (*RISC*), making it ideal for early stage development phases. Tad's contains only the very basics of the *PAT*.

Tad Ray is an extension of Tad with an extra cluster called Ray. Ray is a very simple cluster (one *issue slot*) which provides a Master Interface and some small arithmetic and logic operations.

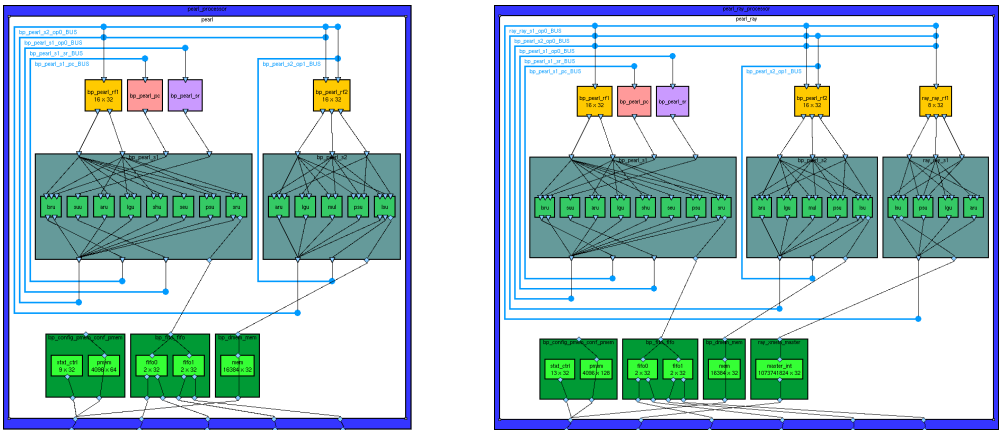
Pearl / Pearl Ray Family

After Tad, Pearl is the next in succession on order of complexity. Pearl is a processor with two *issue slot*s and Pearl Ray is an extension using the cluster Ray.



(a) Tad (b) Tad Ray

Figure 6.1: The Tad core family



(a) Pearl (b) Pearl Ray

Figure 6.2: The Pearl core family

Benchmark	Kind	Description
2mm	Linear Algebra	2 matrix multiplication (D=A.B; E=C.D)
3mm	Linear Algebra	3 matrix multiplication (E=A.B; F=C.D; G=E.F)
adi	Stencil	Alternating direction implicit solver
atax	Linear Algebra	Matrix transpose and vector multiplication
bicg	Linear Algebra	BiCG sub kernel of BiCGStab Linear Solver
covariance	Datamining	Covariance computation
doitgen	Linear Algebra	Multiresolution analysis kernel
durbin	Linear Algebra	Toeplitz system solver
fdtd-2d	Stencil	2-D finite different time domain kernel
fdtd apml	Stencil	FDTD w/ Anisotropic Perfectly Matched Layer
floyd warshall	Graph Analysis	Floyd-Warshall Algorithm
gemm	Linear Algebra	Matrix multiply C=alpha.A.B + beta.C
gemver	Linear Algebra	Vector multiplication and matrix addition
gesummv	Linear Algebra	Scalar, vector and matrix multiplication
jacobi 1d	Stencil	1-D Jacobi stencil computation
jacobi 2d	Stencil	2-D Jacobi stencil computation
lu	Linear Algebra	LU decomposition
ludcmp	Linear Algebra	LU decomposition
mvt	Linear Algebra	Matrix vector product and transpose
reg detect	Image Processing	2-D image processing
symm	Linear Algebra	Symmetric matrix multiply
trisolv	Linear Algebra	Triangular solver
trmm	Linear Algebra	Triangular matrix multiply

Table 6.1: Table of benchmarks used for validation and performance comparison. Source: [5]

Cec 2is / Cec 4is / Cec 8is Family

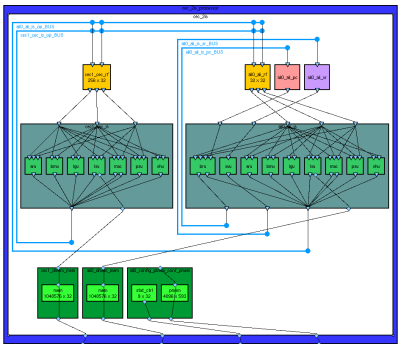
Although not part of the regressions, these three architectures were already made public in another Master's Thesis [23], making them also good candidates for the purpose of validation and benchmarking.

Cec 2is, Cec 4is and Cec 8is have respectively 2, 4 and 8 issue slots. All the architectures have one common control slot and the remaining ones contain both arithmetic, logic and load-store operations.

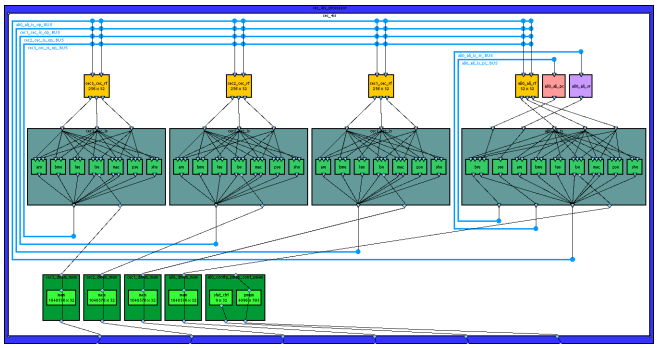
Avispa demo 1

The last processor of this set is the *Avispa Demo 1*. This core was once part of the demo suite used in the early days of ICG's technology, formerly called Silicon Hive. Avispa Demo 1 is still currently used internally for trainings with the tool flow.

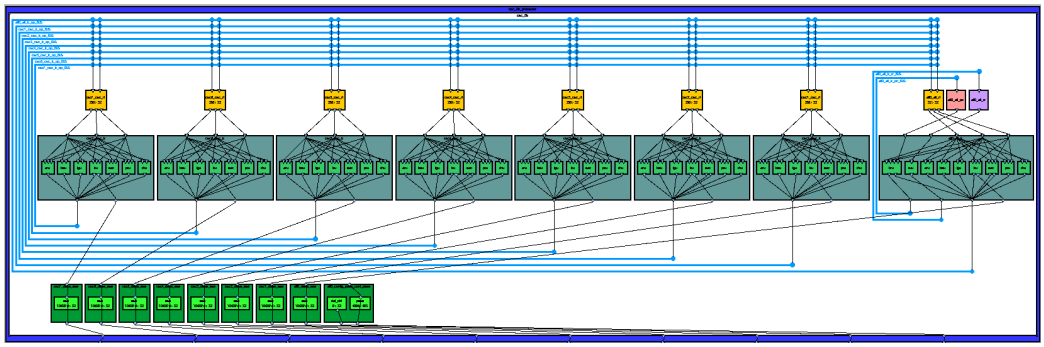
Avispa is one of the most complete architectures of the regression set in the amount of features of the *PAT* it implements but also is one of the heaviest, with an outstanding number of 26 *issue slot*s and 35 register files, making it a good candidate for validation and benchmarking purposes.



(a) Cec 2is



(b) Cec 4is



(c) Cec 8is

Figure 6.3: The Cec Processor family

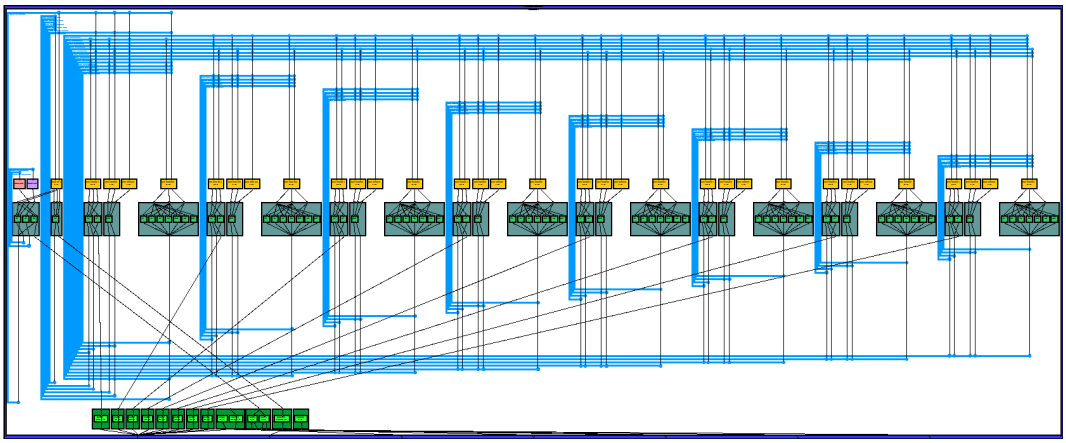


Figure 6.4: Avispa Demo 1 Processor

6.3 Validation

The first step for validating the proposed solution was verifying its correctness when compared to current cycle-accurate simulation framework.

Validation was performed with the presented core set against all the presented reference benchmarks.

Several sizes of data input were used as well and all three methods to validate solution

Core	Instruction Word Size	Number Slots	Average operations/slot	Number Register Files
Tad	32	1	56	3
Tad Ray	64	2	48	4
Pearl	61	2	55	4
Pearl Ray	111	3	50	5
Cec 2is	112	2	95	4
Cec 4is	246	4	91	6
Cec 8is	522	8	89	10
Avispa Demo 1	900	26	27	35

Table 6.2: Metrics on the core set

accuracy. The only exception was for benchmarks done with heavy datasets, where the trace output of the simulation became too big for the environment to handle, so validation was only performed by comparing execution number of cycles, final memory and register contents.

6.4 Performance

6.4.1 Initial Results

As previously mentioned in Section 5.3, the proposed solution did not fulfil performance requirements in early stages of development. These initial results can been seen in Figure 6.5.

Core	Compiled (s)	ISS (s)
Tad	39.8	21.3
Tad Ray	28.1	17.2
Pearl	147.2	66.23
Pearl Ray	145.2	772.8
Cec 2is	88.7	397.1
Cec 4is	52.6	405.2
Cec 8is	54.48	675.1
Avispa Demo 1	55.4	1916.2

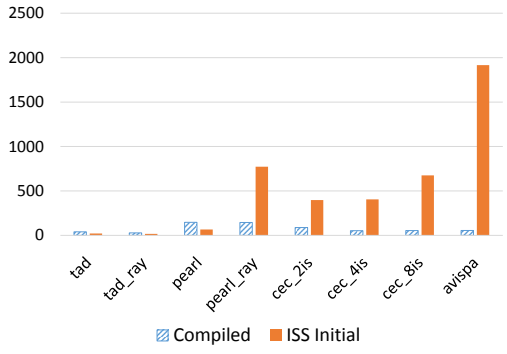


Figure 6.5: Initial results without any performance optimizations. The values are the result of the geometric mean of simulation times in seconds for the given benchmarks.

In this figure, it is very clear that the performance degrades significantly for architectures with instruction word sizes bigger than 64 bits. This is directly related to the decoding effort of every cycle, as the lookup table optimization significantly reduced simulation times.

6.4.2 Pre-decoded Lookup Table Optimization

The optimization results of using lookup tables instead of on-the-fly decoding can be seen, as a ratio compared to its non-predecoded version, in Figure 6.6. The ISS represents the cycle accurate version of the simulator while *ISS 32* represents the decoupled version with *N=32*.

Core	ISS	ISS32
Tad	1.16	2.17
Tad Ray	1.21	2.06
Pearl	1.29	1.56
Pearl Ray	12.66	42.46
Cec 2is	7.96	33.59
Cec 4is	13.00	34.81
Cec 8is	17.81	33.63
Avispa Demo 1	19.03	28.09

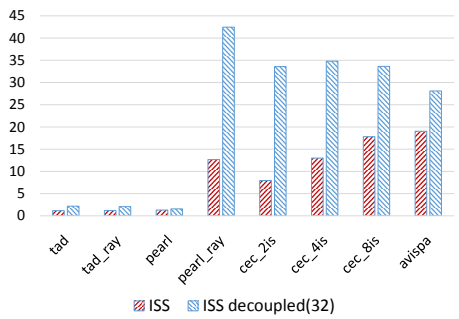


Figure 6.6: Predecode lookup tables performance improvements. The values are the result the geometric mean of simulation times for the given benchmarks, normalized to the not pre-decoded versions' results.

6.4.3 Simulation Time Decoupling Optimization

The second optimization performed in the *Instruction Set Simulator* model was time decoupling from the scheduler. Specially for small architectures, where the time taken to execute one cycle is smaller than in bigger ones, the time spent on the simulation scheduler was significant, making time-decoupling having a significant impact on simulation time.

In cases where the effort to run a single cycle is considerably heavy, like in architectures with big instruction words and no pre-decoding optimization, time decoupling does not provide a significant improvement of simulation time.

The complete results for the time-decoupling optimization can be seen in Figure 6.7.

Core	Pre (s)	No Pre (s)
Tad	7.89	4.23
Tad Ray	5.99	3.53
Pearl	4.25	3.54
Pearl Ray	3.66	1.09
Cec 2is	4.50	1.07
Cec 4is	2.79	1.04
Cec 8is	1.98	1.05
Avispa Demo 1	1.53	1.04

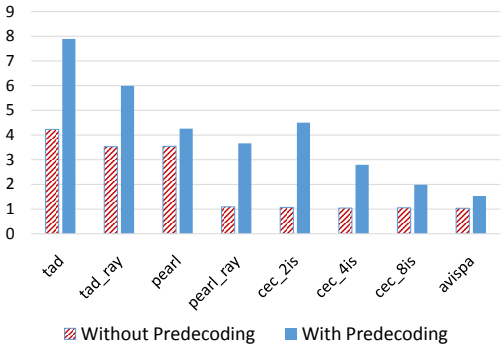


Figure 6.7: Performance improvements by decoupling cycle-accuracy from the scheduler. The values are the result the geometric mean of simulation times for the given benchmarks, normalized to the not time-decoupled versions' results.

6.4.4 Final Results

The final results of the development of the proposed solution, with lookup table optimizations, can be seen in Figure 6.8.

Core	Comp	ISS	ISS32
Tad	39.8	18.4	2.3
Tad Ray	28.1	14.1	2.4
Pearl	147.2	51.0	12.0
Pearl Ray	145.2	61.0	16.7
Cec 2is	88.7	49.9	11.1
Cec 4is	52.6	31.2	11.2
Cec 8is	54.48	37.9	19.1
Avispa Demo 1	55.4	100.7	65.9

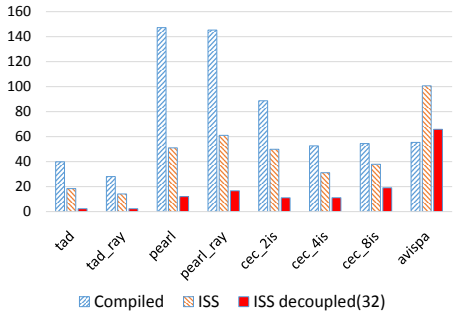


Figure 6.8: Final Performance Benchmarks. The values are the result of the geometric mean of simulation times in seconds for the given benchmarks and cores.

6.5 Overhead

This section presents the overhead of re-generating a new *Instruction Set Simulator* for every new architecture while doing design space exploration.

The overhead time for the generation of a new *Instruction Set Simulator* can be found in Table 6.3, where generation time represents the time spent by the *Instruction Set Simulator* code generator itself and Compile Time the time *GCC/G++* takes to compile the generated code into a simulation model executable.

Core	Number Slots	Instruction Word Size	Generation Time	Compile Time
Tad	1	32	0.74s	5.71s
Tad Ray	2	64	1.12s	7.10s
Pearl	2	61	1.16s	11.04s
Pearl Ray	3	111	1.15s	13.16s
Cec 2is	2	112	1.17s	10.22s
Cec 4is	4	246	2.99s	15.18s
Cec 8is	8	522	6.10	26.87s
Avispa Demo 1	26	900	7.17s	53.51s

Table 6.3: Core, core sizes, generation and compile times of ISS for the core set using -O3 aggressive optimizations

6.6 Discussion

The results obtained for the performance of the proposed solution were unexpected, as performance was expected to always be worse against the compiled simulator. The following discussion tries to justify why this happened.

6.6.1 In-depth comparison of Compiled Simulator vs ISS

Input Files

As mentioned in the background, the compiled simulator approach currently accepts as an input an internal representation file. This internal representation file is in fact a data-flow like description of variables, the registers where they are stored, what operations are executed at which cycles, etc.

The data-flow approach allows a quicker design space exploration but backfires as the simulator, originally designed as a functional simulator, now needs to provide cycle-accurate behaviour so processors can be validated against it.

To ensure that the simulation behaviour is the same as in the processor architectures, the compiled simulator checks constantly at run-time for violations in the pipeline stages (needs to make sure a multi-cycle actually terminates) and write-back data-path to make sure the original *internal representation* can actually be executed in a processor without violating hardware restrictions.

In the *Instruction Set Simulator*, violations of pipeline stages can never happen (when a new multi-cycle operation is started, it continues its execution naturally) and the data-path is already modelled as part of the simulation flow and can be checked at compile time, instead of run-time.

Operation Semantics

Another big difference from the compiled simulator to the proposed *Instruction Set Simulator* is how the operation semantics are handled. In the compiled simulator, to accelerate re-compilation of a simulation model, the semantics are already compiled in a separate library that is linked dynamically after compiling the main simulation model.

This accelerates re-generation of a simulation model but powerful compile-time optimizations such as in-lining cannot be fully used. As the proposed *Instruction Set Simulator* is compiled only once, compile time can be sacrificed and the semantics can be compiled together with the simulation model, allowing more powerful optimizations such as in-lining or constant propagation.

6.6.2 Small and large architectures

As one can see clearly by the results, the bigger the architecture the slower the simulation. This has naturally to do with both the model code size and the amount of control data that a bigger architecture demands.

Even with the pre-decoding optimization, there is still a need to verify and execute every *opcode* and as more *issue slot*s an architecture has, more checks need to be done, resulting in higher simulation times.

This represents a major disadvantage versus the compiled simulator when running low-density code. By low-density code it's meant a program where very little of the issue slots potential is actually used. This is easily encountered when code is not optimized or parallelized properly beforehand.

In this cases, because the compiled simulator simply needs to simulate the code that actually exists, its execution time becomes proportional to the amount of code it needs to

simulate. In an Interpretative *Instruction Set Simulator* every simulation cycle an instruction needs to be fetched and executed, even if it is simply a big set of *NOP* operations, making its simulation time grow with the size of the architecture.

6.6.3 Generation time of the ISS

Discussed previously, it is important that the time to re-generate an *Instruction Set Simulator* does not become cumbersome. From the results shown in 6.3, even for the bigger architecture *Avispa*, the complete time to regenerate a new simulation model is of about one minute.

Having in mind that this time is unique for every architecture and should be put together with the time taken by the other tools to re-generate the tool set for a new architecture, the impact on the user is practically insignificant.

Chapter 7

Conclusions and future work

7.1 Conclusions

This work began by presenting the ICG processor development flow, its features and some of the problems within its simulation framework. To address these, some requirements for a new simulation framework were established and was done a discussion on the challenges that having such requirements created.

Section 4.2 defined a set of requirements: *compatibility with current framework*, *binary transparency*, *modularity*, *human readability* and finally having a possible *integration into other simulation frameworks*. The following paragraphs will go over this requirements one last time and discuss their fulfilment in this work.

Compatibility with current framework

This was one of the most challenging requirements that this work had to address as the solution had to be retargetable for any processor architecture described within ICG's Processor Architecture Template. To address this requirement, a tool to generate a new Interpretative Instruction Set Simulator model for each possible core was described in Section 5.2.

Binary transparency

The requirement of binary transparency was created to directly address ICG's need for a binary validation mechanism. To fulfil this, the tool presented in Section 5.2 implemented the same Instruction Set Architecture as the one described in the input Processor Description File.

Modularity and Human Readability

This requirement was created to allow better maintainability and possible extendibility of future generated simulation models. Modularity was addressed by separating the simulation models into smaller, self-contained modules (c++ classes). Human Readability was addressed by replicating the nomenclature (names of components, connections, etc) inside the generated simulation models.

Integration into Simulation Frameworks

Integration into Simulation Frameworks was created to allow the use of the solution into any industry simulation framework. Because SystemC has become the de facto simulation language in recent years, it is hard to find an industry simulator that does

not implement some sort of SystemC API that allows the integration of these models. The solution tries to fulfil this requirement by implementing its output in this language.

In response to these requirements, Section 4.3 presented a few challenges to the realization of the solution. From these challenges, targeting a Retargetable Environment was the most difficult one to address. During the implementation of the solution was common to step into overlooked features when switching from one processor architecture to another, but in the end being able to fulfil it was also the most rewarding.

Another challenge worth mentioning is performance. As Interpretative ISSs are expected to be slower to Compiled ISSs, the performance of the new simulation environment was expected to be slower at all times, when compared to the existing one. To address that, significant performance improvements were added to the final solution.

As discussed in Chapter 6, the final solution presents better simulation performance for small / medium sized VLIW processor architectures but worse when used for larger ones. This allows to conclude that the presented solution in this work is not as scalable as the current simulation framework. Section 6.6 discusses this in detail, but the main issue with the Interpretative was the overhead of having to interpret operations every cycle, while the Compiled approach can optimize this at compile-time.

7.2 Future Work

The simulation models generated by the presented tool are able to simulate any architecture described by the *PAT*. However, there is room to add software development features to assist a software/firmware developer in his tasks. Features such as target code debugging, single stepping and simulation check-pointing support can be added without big development effort and have the potential to significantly improve the work performance of said developers by reducing their time wasted on debugging code.

In a longer term and as a much more complex improvement, native simulation of the binaries through *Just In Time* translation could be interesting to explore, as this is known to be the fastest binary-transparent simulation method currently in the academia and industry. However, applicability to the *PAT* would have to be investigated first. The biggest challenge to the implementation of this feature has to do with the registers used in these architectures. Because the number and size of register files is not limited in the *PAT*, for large architectures it is not possible to directly map a target register to a host register when running native code, which may cause register spilling to be one of the biggest challenges on the implementation of this feature.

Bibliography

[1] Intel. Intel internal documentation, 2014.

[2] IEEE Computer Society. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std. 1666-2011*, January 2012.

[3] Cadence. *Xtensa Processor Developers Toolkit*, 2014.

[4] Cadence. *Tensilica Software Development Toolkit (SDK)*, 2014.

[5] Louis-Nol Pouchet. Polybench overview, 2015. http://www.cs.ucla.edu/~pouchet/software/polybench.

[6] Jianwen Zhu and D.D. Gajski. An ultra-fast instruction set simulator. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(3):363–373, June 2002.

[7] Hao Liu et al. Research and implementation of random test generator for vliw dsps. In *Third International Conference on Information Science and Technology*, 2013.

[8] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA '83 Proceedings of the 10th annual international symposium on Computer architecture*, 1983.

[9] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Vissers, E.J.D. Pol, M.I.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken. Trimedia cpu64 architecture. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 586–592, 1999.

[10] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, 2000.

[11] Accelera Systems Iniciative. About systemc, 2015. http://www.accellera.org/community/systemc/about_systemc/.

[12] Candence. Tensilica-ip overview, 2015. http://ip.cadence.com/ipportfolio/tensilica-ip.

[13] Candence. Xtensa customizable processors, 2015. http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[14] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren. A methodology and tool suite for c compiler generation from adl processor models. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1276–1281 Vol.2, Feb 2004.

[15] Zhe Zhang, Xiaoming Hu, and Linxiang Shi. High-performance instruction-set simulator for tms320c62x dsp. In *Industrial Mechatronics and Automation (ICIMA), 2010 2nd International Conference on*, volume 1, pages 517–520, May 2010.

[16] M. Hamayun, F. Petrot, and N. Fournel. Native simulation of complex vliw instruction sets using static binary translation and hardware-assisted virtualization. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 576–581, Jan 2013.

[17] Li Kong, Haoshan Shi, Jing Liu, Jian Wu, Lei Deng, Zhenjiang Wang, and Fangquan Lin. Efficient optimizations for instruction set simulation. In *Automatic Control and Artificial Intelligence (ACAI 2012), International Conference on*, pages 963–966, March 2012.

[18] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 13–18, Oct 2003.

[19] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(12):1625–1639, Dec 2004.

[20] U. Kuhne, S. Beyer, and C. Pichler. Generating an efficient instruction set simulator from a complete property suite. In *Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on*, pages 109–115, June 2009.

[21] Wenhao Jia et al. Mrpb: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, 2014.

[22] et al Dheeraj D. Optimization of automatic conversion of serial c to parallel openmp. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on*, 2012.

[23] Felipe Augusto Chies. Validation and evaluation of the asam. Master's thesis, Universidade Federal do Rio Grande do Sul, 2013.

[24] WindRiver. Simics, 2015. http://www.windriver.com/simics/.

[25] WindRiver. Simics overview, 2015. http://www.windriver.com/products/product-overviews/simics_po_0520.pdf.

[26] WindRiver. *Simics Reference Manual*, 2014.

[27] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Design Automation Conference, 2003. Proceedings*, pages 758–763, June 2003.

[28] I. Oussorov, W. Raab, U. Hachmann, and A. Kravtsov. Integration of instruction set simulators into systemc high level models. In *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, pages 126–129, 2002.